BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

(4)

Report No. 6991

Contract N00014-87-C-0085

INTEGRATION OF SPEECH AND NATURAL LANGUAGE FINAL REPORT

S. Boisen, Y. Chow, A. Haas, R. Ingria, S. Roukos, R. Scha, D. Stallard, M. Vilain

March 1989

Prepared by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, Massachusetts 02238

Prepared for:

DARPA

Report No. 6991

INTEGRATION OF SPEECH AND NATURAL LANGUAGE FINAL REPORT

S. Boisen, Y. Chow, A. Haas, R. Ingria, S. Roukos, R. Scha, D. Stallard, M. Vilain

March 1989

Prepared by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, Massachusetts 02238

Prepared for:

DARPA

Table of Contents

List of Figures

Executive Summary

This report describes the progress during both years of the project, from January 1, 1987 to December 30, 1988. During the course of the project, we have focused on two major activities:

- Developing syntactic and semantic components for natural language processing.
- Integrating the developed syntax and semantics with speech for speech understanding.

To measure the coverage of the syntactic and semantic components and the performance of the integrated system, we use the DARPA 1000-word Resource Management Database corpus. This corpus has been used for developing a standard speech database for evaluating the performance of speech recognition algorithms developed under the Strategic Computing Program [40].

Development and evaluation of the natural language processing components—i.e. syntax and semantics—are done using a training corpus and a test corpus, both drawn from the Resource Management domain. The training corpus is examined by the system developers and is used to determine the phenomena that should be handled. The test set is never examined by the developers; its purpose is to allow us to estimate performance on an independent set that would be representative of the ultimate system's performance in the field.

Our work on natural language syntax included the developement of a broad coverage grammar in an augmented context free formalism—The BBN Annotated Context Free Grammar (BBN ACFG). The grammar formalism, the rules, and the parsing algorithm are described in more detail in Chapter 2. The syntactic phenomena that the grammar handles are also described. Currently, the grammar covers 94% of the training corpus and 88% of the test set. The parsing algorithm extends the Graham, Harrison, and Ruzzo algorithm [15] to incorporate unification. The new algorithm is reviewed briefly in Section 2.2 of this report and in full in [17].

The semantic component currently applies to the output of the parser to produce representations in a higher-order logical language. The semantic component uses a *multi-level semantics* architecture [7]: semantic processing procedes through several stages, performing the assignment of function-argument relations, lexical disambiguation, and translation into an application oriented representation in discrete steps, rather than in a single, complex translation. We have completed the basic framework of the semantic component, implemented a semantic type system, and have added the semantic information necessary to correctly handle a large number of lexical items. Since we do not have access to the actual Resource Management Database, we have also developed a mock-up database to test the results of the semantic component. Currently, the semantics is able to map 75% of the training sentences and 52% of the test sentences to a lexically disambiguated expression. The figures for mapping to an application oriented representation are 44% for training and 32% for test.

To integrate the natural language components with the speech recognition system, we have developed and implemented a search strategy that integrates the syntax, semantics, and hidden Markov word models into an algorithm that finds the best interpretation of the input speech. The search therefore integrates natural language

knowledge sources and uses their constraints to find the word transcription and meaning of input speech. The search is based on the parsing algorithm that has been developed for the syntax, and applies semantics as a post process. The parser is used to find a set of grammatical sentences that have a high acoustic likelihood score given the speech. This set of sentences is ordered by decreasing likelihood. The semantic component is then applied as a post process on this set of sentences to determine the highest scoring meaningful utterance. That sentence is the recognized sentence. The search is currently implemented and has been tested on the standard DARPA Resource Management speech database. Results, in terms of number of words correctly recognized, are: 84.55% without any natural language information; 92.5% with syntax alone; 93.1% with syntax and semantics. In terms of sentences correctly recognized, the results are: 27.7% without any natural language information; 62% with syntax alone; 63.6% with syntax and semantics.

In addition to the above accomplishments, we have also demonstrated the serial connection of speech recognition with the natural language component. In this case, the speech uses a language model that is different from the syntactic and semantic components of the natural language processing system. This approach is not optimal and we think it is only applicable for applications of very low perplexity (less than 50). The integrated approach described above is required for larger perplexity tasks. Nevertheless, the serial demo is useful to demonstrate speech understanding in an actual task, due to its speed in recognition. The speech component had a vocabulary of 600 words and a finite state grammar with a perplexity of 40. The recognized word string was passed to the natural language component which interpreted the request, accessed the database system, and presented the output using the a simulation of the OSGP graphics system. This demonstration was capable of handling a demo scenario but was not considered a robust system. The serial connection was demonstrated on two occasions: once to the program manager and once to all participants of the Strategic Computing Speech Program meeting on October 13--15, 1987.

The structure of the rest of this report is as follows. Chapter 1 provides a general overview of the architecture of the system. Chapter 2 describes the grammar formalism, the parsing algorithm, and the syntactic phenomena handled by the grammar. Chapter 3 describes the family of logical languages that are used in the semantic component, the semantic processing framework, and the way in which the syntactic phenomena described in Chapter 2 are interpreted by the semantic component. Chapter 4 describes the techniques that we have developed for integrating speech and natural language over the last two years, as well as the actual speech processing algorithms used. Chapter 5 gives a list of the papers published by the members of the project over the course of the last two years, as well as oral presentations and conference submissions that have been accepted for presentation.

1. System Overview

This chapter provides a general overview of the BBN Spoken Language System. It describes, in broad terms, the syntactic component (Section 1.1), the semantic component (Section 1.2), and the speech algorithms and speech and natural language integration work (Section 1.3). It also describes the techniques that have been used for developing and evaluating the natural language components (Section 1.4) and talks about the implementations of the system that have been developed (Section 1.5).

1.1 The Syntactic Component

The syntactic component of the BBN Spoken Language System uses a declarative grammar formalism—The BBN ACFG (for Annotated Context Free Grammar)—that utilizes complex symbols in its rules, rather than atomic symbols, as ordinary context-free grammars do. Section 2.1 describes this formalism in more detail; for the purposes of the present overview discussion, we will consider a typical rule from the grammar to illustrate the general features of the formalism as a whole. While a rule stating that a sentence (**S**) is made up of a noun phrase (**NP**) followed by a verb phrase (**VP**) and an optional sentential adjunct (**OPTSADJUNCT**) would be written as follows in an ordinary context-free grammar:

S → NP VP OPTSADJUNCT

in the BBN ACFG, the corresponding rule is as follows (omitting details not relevant for this example):

```
(S ... :MOOD (WH-) ...) → (NP :AGR ...)
                          (VP :AGR :MOOD ...)
                          (OPTSADJUNCT ...))
```

There are several aspects of this rule worth pointing out. First, the atomic symbols of the corresponding context-free rule are replaced by complex symbols, represented here (and in the actual grammar) as lists. Each category of the grammar takes a fixed number of arguments, in a fixed order. Each argument, in turn, may take on one from a fixed set of values. Values are either constants (represented as lists, such as **(WH-)**) or variables (represented as symbols with a leading colon, such as **:AGR**). This rule states that a declarative (**(WH-)**) sentence consists of a noun phrase followed by a verb phrase and an optional adjunct. The use of the **:AGR** variable in the NP and VP elements of the rule enforces agreement of the NP and VP in person and number. Similarly, the **:MOOD** variable in the S and VP elements requires that they have the same mood.

This rule illustrates the use of variables in different elements of a rule to enforce agreement among those elements. This is a standard use of variables in complex feature based grammars. The BBN ACFG also uses variables to link dislocated elements to the position in which they receive their semantic interpretation; for example, in a sentence such as "Who did John see?", "who" must be interpreted as the direct object of "see". This use of variables is based on the mechanism of difference lists introduced by Pereira [37], and is described in more detail in Section 2.1.4.

In addition to the BBN ACFG, there are currently many different complex feature based formalisms. The BBN ACFG grammar is a relatively "stripped down" version of such a grammar; it does not support the use of the relatively sophisticated mechanisms that are part of other formalisms, such as feature (argument) disjunction, feature negation, metarules, optional arguments, and the use of attribute-value pairs, rather than positional arguments. We have found this Spartan character to actually be of great use in writing a large (over 800 rules) grammar, while still maintaining consistency. Section 2.1.5 discusses this issue in more detail and compares the BBN ACFG formalism with other annotated context-free formalisms.

The parsing algorithm used to parse the BBN ACFG is based on the Graham, Harrison, Ruzzo (GHR) algorithm [15], iself an extension of the familiar Cocke-Kasami-Younger (CKY) algorithm for context-free grammars. Our algorithm is, in turn, an extension of the GHR algorithm to support complex feature-based grammars—the original GHR algorithm only handles true context-free grammars; Section 2.2 provides more details. The algorithm is guaranteed to find all the parses for an input sentence. We have chosen such an "all-paths" parsing algorithm, rather than a deterministic algorithm as advocated by Marcus [32], for a number of reasons: (1) for the spoken language system application, an all-paths approach is crucial, since the input is indeterminate; one needs to find all the possible ways of interpreting the input and rank them; and (2) even for typed input, which does not have the problems of speech, it is not clear that the deterministic approach is correct; there are truly ambiguous utterances, such as the familiar "Flying airplanes can be dangerous"; advocates of the deterministic approach have not specified how such cases of ambiguity are to be handled.

On the other hand, the all-paths approach can occasionally lead to extremely large numbers of parses for a given sentence, particularly in cases involving conjunction and prepositional phrase attachment. To deal with this issues, as well as to extend syntactic coverage to include common phenomena that increase ambiguity—such as "the deletion": "Show the chart of Mozambique Channel with the Davidson displayed *in center*"—we have experimented with assigning scores to rules. We have performed two experiments along this line; in the first, scores were assigned by hand; in the other probabilities were assigning automatically. The results have been encouraging; see Section 2.6.4 for more details

The current ACFG grammar contains 866 rules; of these, 424 introduce grammatical formatives (such as the articles "a", "the", prepositions, etc). The remaining rules handle the general syntactic constructions of English. Coverage on the training corpus is currently 91% and coverage of the test corpus is 81% with this grammar. The version of the grammar used by the parser that utilizes rules with scores contains 873 rules. Coverage with this version of the grammar is 94% on training and 88% on test.

1.2 The Semantic Component

Once syntactic processing is done, the resulting syntactic structure is mapped into a corresponding semantic representation. While it might seem that all that is necessary is a simple mapping from a syntactic representation to a semantic representation, and from the semantic representation, perhaps, a mapping to the final application, there is evidence, beginning at least with work on the PHLIQA1 system ([7]), that semantic interpretation should be a multi-level process. In this section, the architecture of the multi-level semantics component of the BBN Spoken Language System is sketched out, along with some examples of semantic analyses in our system.

Figure 1-1 shows the general architecture of the natural language processing components of the BBN Spoken Language System. A word sequence is input to the syntactic processor (parser) which uses the grammar to assign one or more parse trees to the sequence. Each parse tree produced is then passed to the first stage of semantic interpretation. This interpreter, using the semantics (typically stored in the lexicon) associated with each terminal (lexical item or grammatical formative), as well as the semantic formation rules that assign an interpretation to each grammar rule, produces one semantic representation for each syntactic parse tree. The logical language in which these semantic representations are expressed is known as EFL (for English-oriented Formal Language). EFL is a level at which function-argument relations are correctly stated. However, EFL is a level that can contain ambiguous logical constants. For example, the English word "bank", which is ambiguous in meaning between a financial institution and the side of a river, might have an ambiguous EFL constant. Ambiguous constants at this level are resolved in the translation to the next logical level, WML (for World Model Language). In the translation from EFL to WML, domain specific information (which is usually stored in a "domain model") is used to disambiguate ambiguous EFL constants, where possible. It should be noted that while the translation from syntactic parse tree to EFL was a one-to-one mapping, the translation from EFL to WML is not so constrained and, for any given utterance, may wind up being many-to-one or one-to-many. It is many-to-one, when a given utterance has more than one parse, but the resulting EFL expressions are logically equivalent and so can be reduced to a single WML expression. It is one-to-many when an EFL expression contains one or more ambiguous constants which cannot be disambiguated in the translation to WML.

Each WML expression is translated into an expression of DBL (for Data Base Language). DBL, as the name implies, is a level of representation that is meant to be fairly close to an expression in an application's interface language. DBL expressions cannot actually be used in the end application, but the translation from DBL to such a language (such as SQL) is straightforward since DBL includes a constant symbol for each data file of the database.

The last stage of processing is *evaluation*, which computes the "value" of a DBL expression against the actual contents of the data files in the database. This value constitutes the answer to the original input question[it is expressed in yet another (much simpler) language, CVL (for Canonical Value Language). CVL has a very small set of logical operators and only numerical and string symbols as constants.

In introducing the multi-level semantics architecture, the example case given as motivating it was that of lexical ambiguity. While lexical disambiguation is one of the prime uses of multi-level semantics, it is by no means

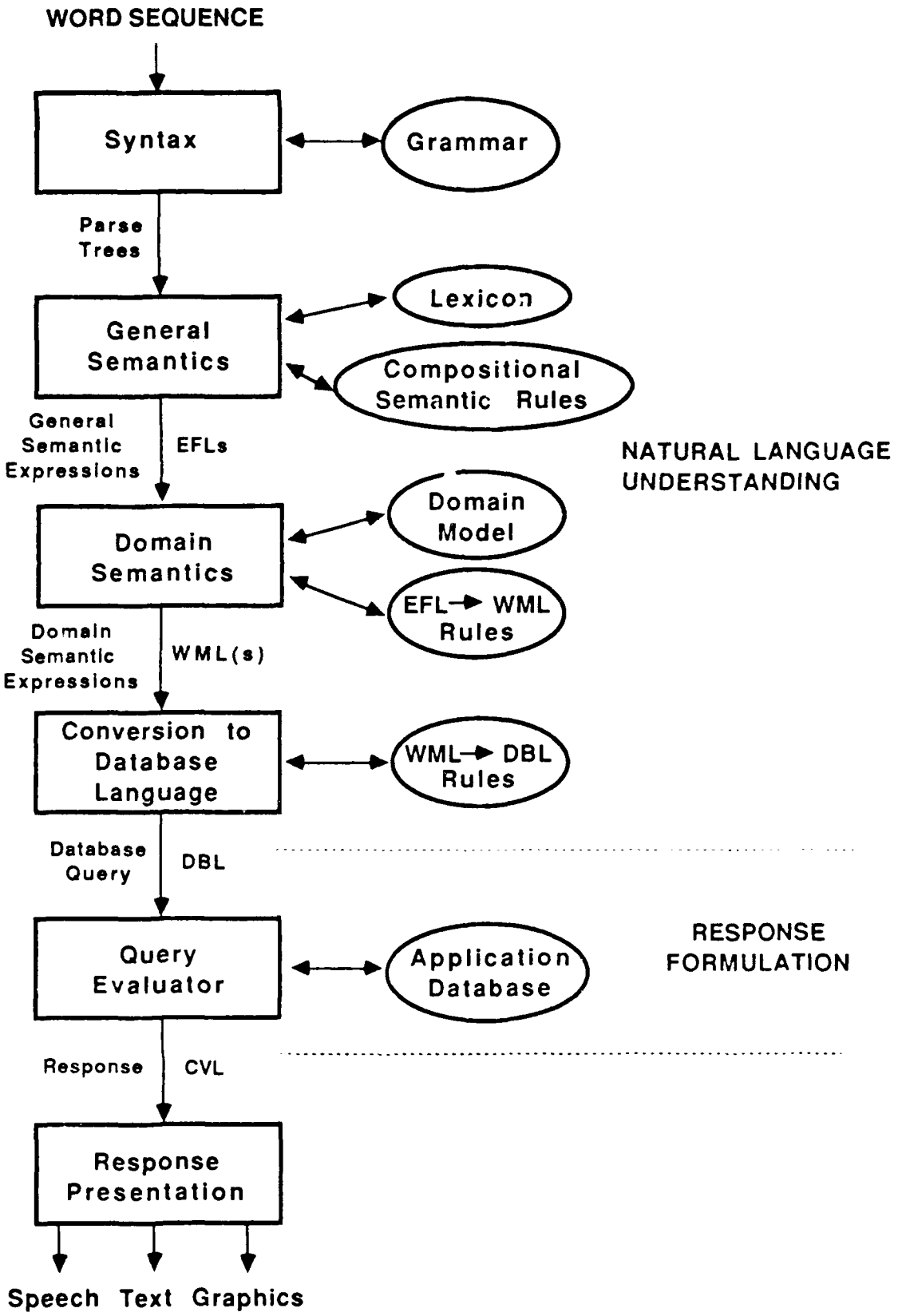


Figure 1-1: Architecture of The Natural Language Processing System

the only use. Three case studies of the use of a multi-level semantics are presented here, to show the general utility of this approach. These examples are all implemented in the BBN Spoken Language System.

Multi-level semantics is used in the interpretation of plural and conjoined NPs [43]. Plural NPs (including conjunctions) can be interpreted either collectively or distributively, depending on the nature of the function with which they are associated. The following examples illustrate this:

Sentence:	**Semantic Representation:**	**Plural Type:**
The boys walk.	$\forall x \in$ BOYS: WALK[X]	Distributive
	WALK[BOYS]	Collective
The boys gather.	GATHER[BOYS]	Collective

In "The boys walk", the property of walking can be predicated of each boy; this sort of plural interpretation is called distributive. The property of walking can also be predicated of the entire set of boys; this type of plural interpretation is called collective. In "The boys gather", on the other hand, "gather" can only be predicated of the entire set of boys, since individual boys cannot gather; i.e. only the collective interpretation is available. Since plural NPs can have either interpretation, depending upon the context, it is necessary to have a semantic interpretation framework that allows the context to determine the interpretation. This is not true of all semantic interpretation systems; for example, Montague's PTQ [35] only predicts the distributed interpretation for conjoined NPs, even though in a sentence like "John and Peter carried a piano upstairs", the collective reading is preferred.

The solution proposed in the framework of multi-level semantics is to allow conjoined NPs to be represented on the EFL level with an ambiguous "union" function which does not force either a distributive or collective interpretation. In the mapping from EFL to WML, this function is disambiguated in a way which is constrained by the interpretation of the associated context. The disambiguation function is flexible enough that in cases with multiple interpretations, each interpretation is obtained. For example, "The juries and committees gather" can either mean that each jury and each committee gathers, each separate from the other; or, that all the juries gather in one place and all the committees gather in another; or, finally, that all the juries and the committees gather in one place. The interpretation mechanism sketched here produces all three readings.

Multi-level semantics is also crucial for the interpretation of relational nouns such as "sister" or "commander", which normally are not used as simple kind terms but rather are used to express relations among objects in the domain [13]. That is, one does not normally speak of "sisters", per se, but rather of "sisters of":

?Mary is a sister.
Mary is John's sister.
Mary is the sister of John.
John has a sister.

Note that the element that is interpreted as the argument (in some sense) of a relational noun like "sister" may be associated with the relational noun either via the possessive ("'s"), "of", or "have". However, these items are not limited to filling in the arguments of relational nouns; they can be freely used with ordinary term nouns, as well:

This is a book.
This is John's book.
This is the book of John.
John has a book.

Since relational nouns behave syntactically very much like term nouns and since the relevant formatives "'s", "of", and "have" can be used freely with term and relational nouns, at EFL, no distinction is made between relational and non-relational nouns and "'s", "of", and "have" are not disambiguated. However, in the translation from EFL to WML, "'s", "of", and "have" are disambiguated between their simple "possessive" interpretation and their use as markers of the argument to a relational noun, on the basis of the noun phrases with which they are combined.

Finally, we have implemented a computational implementation of a mechanism for intra-sentential pronominal reference. The syntactic portion of this treatment uses an algorithm inspired by the indexing mechanism described in [10], and is described in detail in Section 2.4. The important point about the algorithm for the purposes of the present discussion is that it finds all possible antecedents for a pronominal expression. The multi-level semantics architecture is used to choose among antecedents in the following way: for each pronominal expression that has one or more antecedents, a new ambiguous EFL constant is generated "on the fly" with each antecedent as a potential translation. In the translation to WML, the semantic type constraints imposed on the pronoun by the constituent of which it is an argument are matched against the semantic type of each potential antecedent: those that do not match are filtered out. Each pronoun is also allowed to have an extra-sentential antecedent, to allow for discourse mechanisms to select among any competing inter- and intra-sentential antecedents.

These examples should suffice to show that the multi-level approach is of great utility in dealing with varied semantic problems.

Since we do not have access to the actual Oracle Resource Management database, we have built a mockup database to pose queries to, to test the coverage of the system. To simplify matters, we have essentially dispensed with the DBL level, requiring the data file structure of our database to correspond to the logical structure of the WML domain model. This database is represented in the language CVL.

Currently, the semantics is able to map 75% of the training sentences and 52% of the test sentences to a WML expression. The corresponding figures for CVL are 44% for training and 32% for test.

1.3 Speech and Natural Language Integration

Over the two year course of the project, we have tried several approaches to integrating speech and natural language. The first approach, which was displayed in a demonstration context, was what we have called the "serial connection". This approach is schematized in Figure 1-2.

In this architecture, the speech component operates completely independently of the natural language processing modules. The speech recognition system, using its own language model, determines the best word sequence corresponding to the input speech. This is then passed to the syntactic component, which, in turn, passes its output to the semantic module to produce a representation of the meaning of the input. While this system performed reasonably well using a small demo scenario (shown in Figure 1-3, it is clearly not well-suited to the general task of speech understanding. There are two main problems: (1) the speech and natural language components use different language models, creating the possibility that the two models might eventually diverge, with the speech language model accepting sequences not accepted by the natural language component, and vice versa; and (2) since the natural language components apply to the output of the speech component, any aid that the natural language modules might provide to speech recognition is inaccessible. Because of these problems, we have tried to more closely couple the speech and natural language components.

The approach that we have taken is the following. First, a lattice engine computes a very dense word lattice; all words that are plausible acoustically are computed. This word lattice is passed to a parser—the "Lattice Parser"—that uses an algorithm similar to that used in the natural language parser for text input, which takes account of the acoustic likelihood scores that have been computed by the lattice engine, as well as syntactic information, to search for the most likely meaningful sentence as a path through the lattice. This architecture is shown in Figure 1-4.

As the figure shows, semantics is currently applied as a post-process to the output of the Lattice Parser. One of our research tasks in the future is to explore the use of an incremental semantic mechanism that can be used during the course of parsing, whether of text or speech input.

Performance of the integrated spoken language system is measured in terms of number of words correctly recognized. As a baseline figure, the percentage of words correctly recognized by the acoustic recognition portion of the system without any natural language information is 84.55%. With syntax alone, the figure is 92.5%; with syntax and semantics 93.1%. The figures for senteces correctly recognized are 27.7% without any natural language information; with syntax alone, 62%; with syntax and semantics 63.6%.

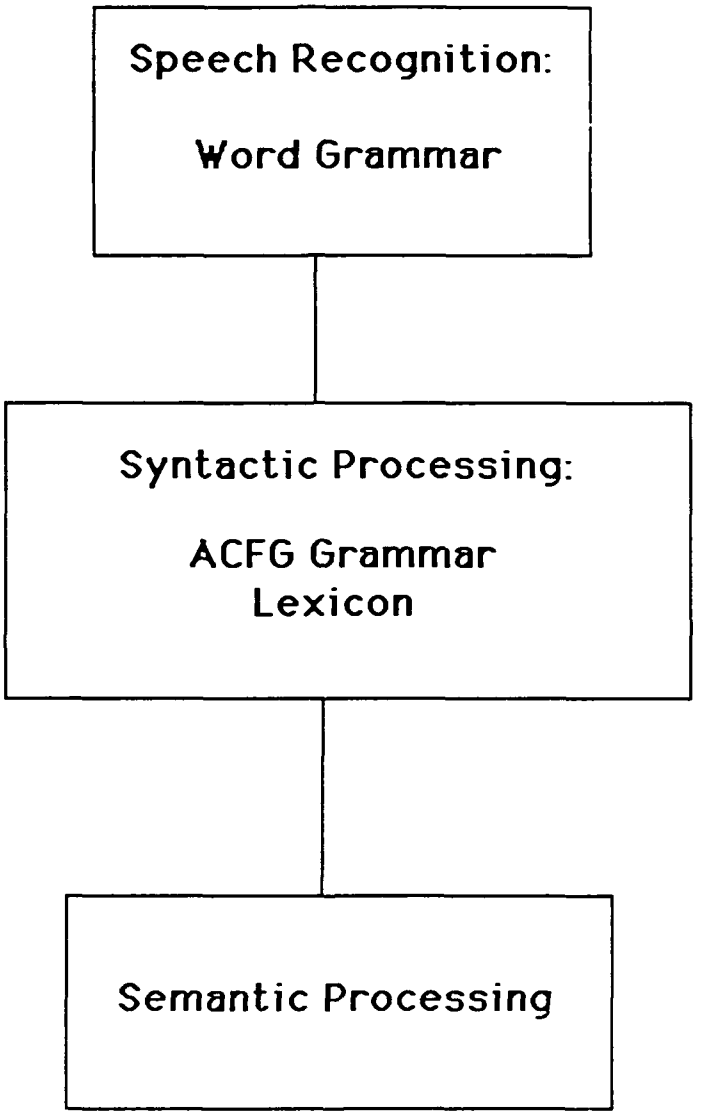


Figure 1-2: Serial Connection of Speech and Natural Language

Show me the Indian Ocean.
Display thirty degrees south seventy degrees.
Where are the frigates and carriers?
What is the readiness of Eisenhower?
When will Eisenhower be C1?
What is Frederick's readiness?
Which ships are faster than Frederick?
Display Frederick's track.

Figure 1-3: Demo Scenario for Serial SLS System

1.4 Development and Evaluation Methodology

In order to see how the work on syntax and semantics that we have done generalizes from the examples that we have seen to new utterances, we have adopted the methodology, now common in speech recognition research, of using both a training set and a test set. Syntactic and semantic work are done on the basis of the training corpus of 791 sentences; the test corpus (200 sentences) is kept hidden from the system developers, to simulate novel utterances that users of the system might make. Periodically, the test corpus is run through the system, again without the system developers looking at any of the sentences. However, statistics are collected on the percentage of sentences parsed; the percentage of sentences that receive well-formed EFL representations; the percentage of sentences that receive well-formed WML representations; and the percentage of sentences that receive answers. These figures can be compared to the corresponding results on the training corpus, to see how well the grammar generalizes from the training (known) to test (unknown) corpus.

These figures give us raw performance figures. In the case of the syntactic component, we can also collect figures on the ambiguity of the grammar. Since the parsing algorithm finds all the parses for a given utterance, a goal of the system is to increase coverage while not explosively expanding ambiguity. This is an especially difficult problem, since the training corpus contains phenomena, such as ellided determiners—e.g. "Show position of Frederick"—that can wildly increase ambiguity if allowed to apply freely. To balance these two goals, we have experimented with versions of the parser in which rules are sorted into different levels of grammaticality. In this version of the parser, parses are ranked according to the rules utilized. We have experimented with two methods of assigning ranks. In the first version, levels of grammaticality were assigned by hand, by the grammar developer. In the second version, the training corpus was parsed, and all the parses produced were cached and used to assign probabilities to each rule.

Initial efforts, in which ranks were assigned to rules by hand, were encouraging. A version of the grammar which included rules such as determiner ellipsis that increased ambiguity, had an average of 18 parses per sentence and a mode of 2 parses. However, when only first order parses were considered the average was 2.86 parses and the mode was 1. The parser without the extra rules and without ranking has an average of 4.26 parses and a mode of 1.

Testing the results of the automatic assignment of rule probabilities against a corpus of 48 sentences (from the

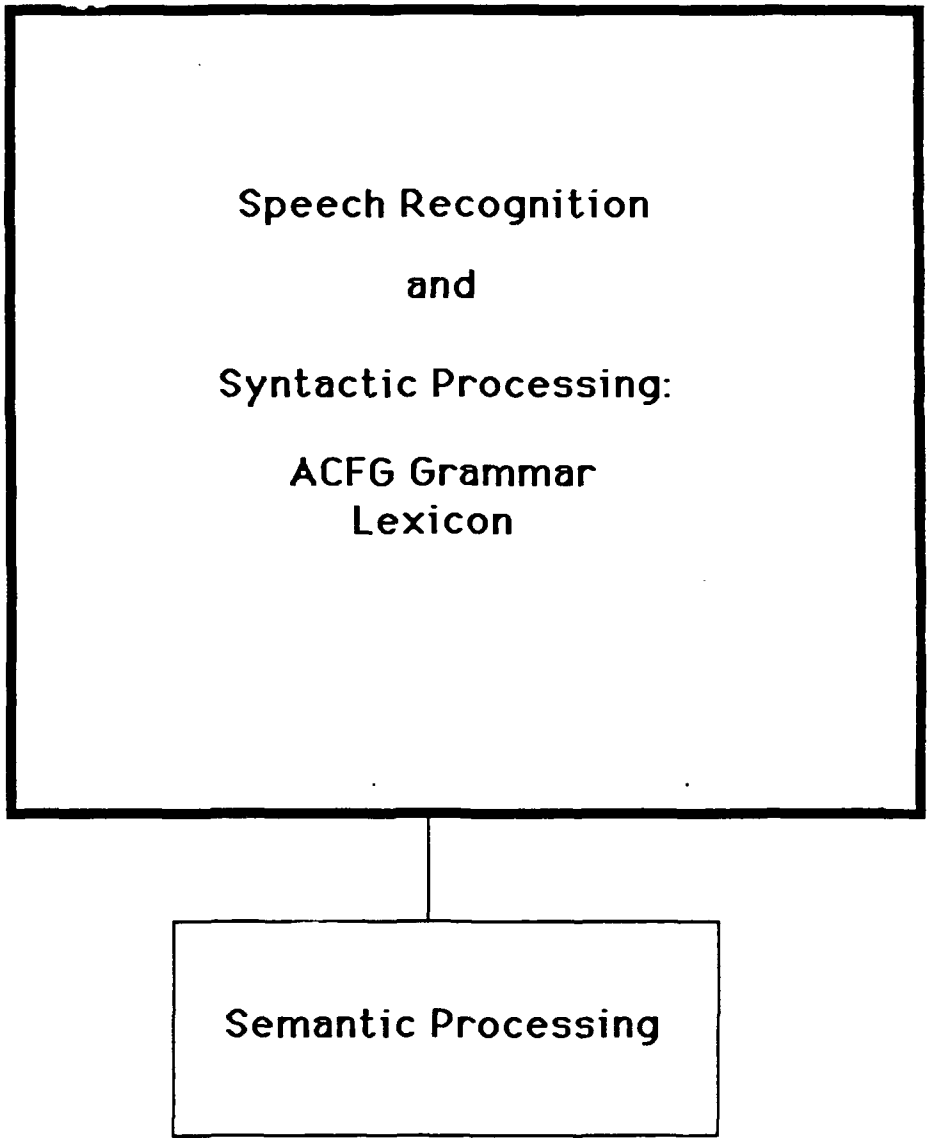


Figure 1-4: Combined Speech and Natural Language Parsing

training corpus) that were manually parsed, the parse assigned the top score by the probabilistic parser was correct 77% of the time. Looking only at the top 6 parses, the correct parse was present 96% of the time. Since the success rate is 96% considering only the top 6 parses, while 50% of the sentences have 6 or more parses, this suggests that this probabalistic approach is on the right track.

1.5 Implementation

The BBN Spoken Language System is implemented in Common LISP. There are currently versions of the system that run on Symbolics 3600-class LISP Machines, TI Explorers, and Sun 3 and Sun 4 workstations running Lucid Common LISP and Franz Allegro Common LISP under UNIX. The main computational core of the system is machine-independent; the idential code runs on all the systems named. A device-independent loader file allows the system code to be run on all these machines; it is conditionalized to handle input/output differences among them. Finally, the Symbolics version includes facilities for displaying the parse trees produced, built on top of the BBN Grapher [48]. This portion of the system makes use of Symbolics ZetaLISP window system code, and is the only non-portable portion of the system.

All Common LISP code—functions, macros, variables, and constants—is interned in the **CFG** package. Throughout this document, any reference to LISP symbols refers to symbols in the **CFG** package, unless otherwise stated.

2. The Syntactic Component

This chapter describes the syntactic component of the BBN Spoken Language System. This component uses a broad-coverage grammar written in an augmented phrase structure grammar formalism and parsed using an algorithm based on the CKY algorithm for context-free grammars. Section 2.1 discusses the format of the syntactic grammar at a relatively high level. Section 2.2 introduces the algorithm that is used to parse the grammar. Section 2.3 describes the implementation of the grammar formalism and parsing algorithm. Section 2.4 describes the mechanism that is used for handling intra-sentential pronominal reference. Sections 2.5 and 2.6 describe the coverage of the grammar, both in descriptive linguistic terms and in more quantitative measures. Section 2.7 describes extensions to the syntactic component that are planned, on the basis of our experiences during the course of this project.

2.1 The Grammar Formalism

The BBN Spoken Language System uses a grammar formalism based on annotated phrase structure rules; this formalism is called the BBN ACFG (for Annotated Context Free Grammar). It is, therefore, in the tradition of augmented phrase structure grammars such as those of Harman [18] and Heidorn [20], [21] and Generalized Phrase Structure Grammar (GPSG) [14], although its immediate inspiration is Definite Clause Grammar (DCG) [36]. In such grammars, rules are made up of elements that are not atomic categories but, rather, are complex symbols consisting of a category label and feature specifications. Rules in the BBN ACFG consist of grammatical symbols—e.g. representing a part of speech, such as **N**, for noun—that take a specified set of arguments (also referred to as *features*). These, in turn, may take arguments of their own. For example, in the current grammar, nouns and verbs contain **AGREEMENT** as an argument. **AGREEMENT**, in turn, takes the arguments **PERSON** and **NUMBER**. Arguments such as **PERSON** and **NUMBER** that do not take arguments but only assume simple feature values can have either constants or variables as values. Variables begin with a colon; constants are unary lists. For example, **PERSON** can take on one of the values **(1ST)**, **(2ND)**, **(3RD)**, and **:P**; **NUMBER**, the values **(SINGULAR)**, **(PLURAL)** and **:N**. Arguments that take arguments of their own, such as **AGREEMENT**, can also take either constants or variables as their values. Again, variables begin with a colon; constants, however, are multiple element lists whose first element is fixed across all values. For **AGREEMENT**, for example, this is **AGR**. Moreover, since the arguments to a feature such as **AGREEMENT** may themselves be either constants or variables, it is possible to have partially specified values for such features. Here are some examples of fully and partially specified arguments for **AGREEMENT**:

```
(AGR (1ST) (PLURAL))
(AGR (3RD) (SINGULAR))
(AGR (2ND) :N)        ;; (number unspecified)
(AGR :P (PLURAL))     ;; (person unspecified)
:AGR                  ;; (agreement completely unspecified)
```

Variables can be used in different elements of a rule as a means of imposing feature agreement. The following rule from the current grammar illustrates this agreement mechanism; features that are relevant to the present discussion are underlined.

```
;; basic toplevel declarative clause rule, ensuring subject-verb agreement
((S (0COMP) :MOOD (WH-) :TRX :TRZ (-CONJ))
 (NP :NSUBCATFRAME :AGR :NPTYPE (-POSS :POSSCLASS) (WH-) (SUBJ) :DET-STATE
     :DEF :DET-CLASS :SONS :TRX :TRY :CONTRACT :CONJC)
 (VP :AGR :NPTYPE :MOOD (AUXV (W :AUX) :NEG) (WH-) :TRY :TRV :CONTRACTX
     :CONJD)
 (OPTSADJUNCT :AGR :SONSX (WH-) :TRV :TRZ))[1]
```

This rule states that a declarative (**(WH-)**)[2] S (sentence) consists of an NP (noun phrase) followed by a VP (verb phrase) and an optional adjunct. The use of the **:AGR** variable in the NP and VP elements of the rule enforces agreement of the NP and VP in person and number. Similarly, the **:NPTYPE** variable requires that the subject NP be of the type selected by the VP (ultimately, by the head verb of the VP). Finally, the **:MOOD** variable in the S and VP elements requires that they have the same mood.

The BBN ACFG grammar is strongly typed. Each grammatical symbol has a fixed number of arguments in a fixed order. Each argument, in turn, has a fixed set of permissible values; this includes both constant values and variable values. While the BBN ACFG is essentially a species of DCG, this type system is an additional restriction, not found in DCGs; this is the essential difference between our formalism and DCGs.[3]

2.1.1 The Relation between The Grammar and the Lexicon

In phrase-structure based formalisms, there is no formally separate lexicon; lexical items are introduced by phrase structure rules just as syntactic categories (''non-terminals'') are. For example, in order for a grammar written in the ACFG formalism to contain the word ''given'', there would need to be a rule of the following sort.

```
((V (NO-CONTRACT) (DITRANSITIVE :PASSIVE) :P :N (EDPARTICIPLE))
 (given))
```

This rule states that ''given'' is a past participle (**(EDPARTICIPLE)**), unspecified for person and number agreement (**:P :N**), that it takes a ditransitive complement structure (**(DITRANSITIVE)**), and that it may appear either in active or passive constructions (**:PASSIVE**). Note that this rule introduces ''given'' in only one of its uses, the ditransitive (as in ''We have given John the book''; ''John was given a book''). There need to be analogous rules for its other uses, as well. While it might be possible to store all these rules, the storage

[1] ACFG rules are represented as LISP lists. The first element of the list corresponds to the left hand side of the rule. The rest of the elements correspond to constituents on the right hand side of the rule.

[2] Read as ''WH minus''; this follows the analysis now standard in generative grammar that declarative clauses bear the feature -WH (''minus WH'') and that interrogative clauses, whether they are content questions or yes-no questions, are +WH (''plus WH''). See Section 2.5.2.1 for more discussion of these two types of question.

[3] An additional difference between our work and standard DCGs is a depth-boundedness restriction, which is discussed in Section 2.2.

requirements for doing so are prohibitive. The number of rules for each lexical item is equal to the number of inflected forms of the item—singular and plural forms for nouns; positive, comparative, and superlative forms for adjectives and adverbs; and all the past, present, and participial forms for verbs—multiplied by the number of subcategorization frames[4] that the lexical item may appear in. Even for a small lexicon, this will result in a large number of rules; for example, for the current Resource Management lexicon, the expansion rate is as in Figure 2-1

Category	Number of Items	Number of Rules
ADV	3	9
V	74	730
ADJ	129	1110
N	577	2222

Figure 2-1: Expansion Figures for Vocabulary Items

For the multi-thousand word lexicons needed for robust natural language processing, there would be an explosion in the number of rules needed. Because of this, the current version of the BBN ACFG does not store rules introducing lexical items, but rather generates them as needed by the parser on the basis of information stored in the lexicon and in conjunction with a morphology program that handles the regularly inflected forms[5]; such rules that are created on demand but not permanently stored are often referred to as "virtual rules". Thus, while the lexicon has no formal place in our system, it is used as a repository of lexical information (subcategorization, semantics, morphology, etc.) that is used to construct the virtual rules that the grammar actually uses.

2.1.1.1 Subcategorization

Virtual rules are also used to ensure that a member of a lexical category (currently V (Verb), N (Noun), and ADJ (ADJective)) appears with the correct *complements*. Complements are so called, in traditional grammar, because they "complete the meaning" of a lexical item in some way. For example, a transitive verb requires a noun phrase to follow it: "John fooled the boys" is grammatical but "*John fooled[6]" is not. Complements are lexically specified in that a given lexical item may or may not require (or permit) a particular category. Thus, intransitive verbs forbid a following noun phrase but may optionally permit other complements; e.g. "*The sun rose the boys" is ungrammatical, but "The sun rose over the mountains" is grammatical, although the phrase "over the mountains" is optional: "The sun rose" is grammatical as well.

The set of complements that a lexical item requires is often referred to as a *subcategorization frame*. While

[4]See Section 2.1.1.1 for discussion of subcategorization.

[5]Irregularly inflected forms are listed in the lexical entry of their base form.

[6]* indicates ungrammaticality.

some formalisms, such as PATR-II [46], place most of the information about subcategorization in the lexicon and contain only a single rule for a category in which a lexical item takes complements—currently these are VP for V, N-BAR for N, and ADJ-BAR for ADJ—the BBN ACFG contains a rule for every subcategorization frame in which a lexical category can appear. Each such rule is "indexed", as it were, by a mnemonically named feature that must appear as the value of the subcategorization feature of any verb that can occur in that frame. (In this, it follows GPSG, which uses a similar indexing scheme.) The following two rules illustrate this aspect of the BBN ACFG formalism:

```
((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (W (W (W (0AUX))))))
     (NOT-NEG)) (WH-) :TRX :TRX :CONTRACTX (-CONJ))
 (V :CONTRACT (INTRANSITIVE :NPTYPE) :P :N :MOOD))

((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (W (W (W (0AUX))))))
     (NOT-NEG)) (WH-) :TRX :TRY :CONTRACTX (-CONJ))
 (V :CONTRACT (TRANSITIVE :NPTYPE (TAKES-ACTIVE)) :P :N :MOOD)
 (NP :NSUBCATFRAME :AGR (REALNP :REAL) (-POSS :POSSCLASS) (WH-)
     (OBJ (AGR :P :N)) :DET-STATE :DEF :DET-CLASS :SONS :TRX :TRY
    :CONTRACT :CONJC))
```

The first rule states that a VP may consist of a V (verb) followed by no other complements if the V bears the feature **(INTRANSITIVE)**. The second rule says that a VP may consist of a V followed by an NP just in case the verb is specified as being **(TRANSITIVE)** and capable of appearing in the active voice (**(TAKES-ACTIVE)**). As is usual, the variables **:P**, **:N**, **:NPTYPE**, and **:MOOD** are used to enforce agreement of the V and VP in these features.

2.1.2 Optional Elements

Currently there are no general mechanisms in the BBN ACFG that permit optional elements or that implement the Kleene star operator, which permits zero or more occurences of a specified element. To implement optionality and Kleene star, special categories are introduced that simulate these operations. By convention, such categories have names that begin with **OPT**. These nodes are used to implement optionality in the following way. First, there is always a rule of the form:

OPT<category> →

that is, a rule that expands to nothing. The inclusion of such a rule is what allows such categories to be optional in the first place. In addition, there will be one or more rules of the form:

OPT<category> → <cat_1> ...

where <cat_1> is a category of the grammar.

The number of such rules and the constituents appearing on the right hand side of the rule determine what type of optionality is being simulated. If there is only one rule, with a single non-optional constituent on the right hand side, there is simple optionality. Pre-verbal ADVPs (ADVerb Phrases) are implemented in this way.

```
((OPTPREVP))
```

```
((OPTPREVP)
 (ADVP :ADVTYPE (VP-INITIAL) (WH-) (NIL) (NIL)))
```

Kleene star is implemented in the following way. Again, there is only a single rule with a single non-optional category on the right hand side, but the rule is recursive, with the constituent appearing on the left hand side also appearing on the right:

OPT<category> → $<cat_1>$ **OPT<category>**

This recursive structure allows there to be multiple occurences of $<cat_1>$. Since **OPT<category>** can always expand as the null string, this recursive rule will bottom out. For example, NPs are allowed to have zero or more positive ADJPs by the pair of rules:

```
((OPTADJP (AGR :P :N) :POSIT))

((OPTADJP (AGR :P :N) :POSIT)
 (ADJP :ADJSUBCATFRAME (AGR :P :N) (POSITIVE :COMPARE) :POSIT (WH-)
       (NIL) (NIL) :CONTRACT :CONJA)
 (OPTADJP (AGR :P :N) :POSIT))
```

Finally, if there is more than one rule of the form:

OPT<category> → $<cat_n>$ **OPT<category>**

where $<cat_n>$ differs in each rule, what is simulated is a rule introducing a set of optional consituents, one for each $<cat_n>$.[7] For example, Ss are allowed to have several different adjuncts by the following set of rules:[8]

```
((OPTSADJUNCT :AGR (NO-SONS) (WH-) (NIL) (NIL)))

;; ADVerbial expressions
((OPTSADJUNCT :AGR (HAS-SONS) :WH :TRX :TRZ)
 (ADVP :ADVTYPE (S-PERIPHERAL (S-FINAL)) :DEGREE :WH (NIL) (NIL))
 (OPTSADJUNCT :AGR :SONS :WH :TRX :TRZ))

;; PP's containing a ''with'' and a predicative phrase ("with switches set
    to off")
((OPTSADJUNCT :AGR (HAS-SONS) :WH :TRX :TRZ)
 (PP (WITHPREP) (NON-PREDICATE-P) (NPPREDOBJECT) :WH :CASEX (NIL) (NIL)
     :CONTRACT :CONJ)
 (OPTSADJUNCT :AGR :SONS :WH :TRX :TRZ))
```

[7]This is not quite true: this formalism actually allows quite a bit more. For example, compare the rule S → ... (CATA) (CATB) (CATC) (where parentheses indicate optionality) with the set of rules S → ... OPTFOO, OPTFOO →, OPTFOO → CATA OPTFOO, OPTFOO → CATB OPTFOO, OPTFOO → CATC OPTFOO. The former treatment restricts the order of CATA, CATB, and CATC, while the latter does not. Morevoer, the latter treatment allows more than one occurence of each of CATA, CATB, and CATC. Thus, this device actually combines Kleene star with an *un*ordered set of optional elements.

[8]This is an *illustrative, rather than an exhaustive*, set of these rules. See Section 2.5.2.4 for a complete list.

2.1.3 "Meta-rules" and Feature Value Default Mechanisms

Some annotated context-free formalisms, such as GPSG, include a mechanism that allows for rules that are, in some sense, predictable variants of other rules to be derived rather than being included in the object grammar. For example, GPSG provides a "meta-rule" facility, which, among other things, provides a means for deriving the passive version of transitive VP rules. The BBN ACFG does not have any such mechanism.[9]

A similar mechanism for compacting the size of grammars is some sort of feature defaulting mechanism, which would allow predictable features of one or more elements of a rule to be left unspecified. Currently, the BBN ACFG provides no such mechanism. It is very unlikely that the grammar formalism will itself provide such a facility, although it might be possible to provide an interface between the rules written and seen by users and developers and the form of the rules used by the parser.

2.1.4 Trace Flags

WH constituents, such as 'who", "what", "how many ships", and "how long" are linked to an empty element (or trace) that appears in the position where the WH constituent is interpreted. For example, in the sentence, "who did John see" "who" is linked to an NP trace in the object position for the verb "see". In English, only one trace may appear in a single clause; compare:

who wonders what John gave Bill t_{what}
*who does Mary wonder what John gave t_{who} t_{what}

In the first case, "who" is interpreted as the subject of the matrix clause and "what" is interpreted as the object of the complement clause, so there is no more than one trace per clause and the restriction is satisfied. In the second case, "who" is linked to the indirect object position of "gave" (indicated by t_{who}) and "what" is linked to its direct object position (indicated by t_{what}), resulting in ungrammaticality. There are various linguistic and computational proposals to enforce this restriction. The one used in the BBN ACFG is that of difference lists, first introduced by Pereira [37].

Each constitutent that may either appear as a trace or contain a trace is given a pair of arguments called **TRACE**. The value of **TRACE** may either be an argument specifying what type of constituent has been moved, along with other salient properties (e.g. for NP, the person and number of the WH consitutent) or **(NIL)**. Constituents that are traces have the trace specification as the value of the first **TRACE** argument and **(NIL)** as the value of the second trace argument. For example, the rule introducing NP traces is:

```
(1) ((NP :NSUBCATFRAME (AGR :P :N) (REALNP :REAL)
         (-POSS (NOT-POSS)) (WH-) :CASEX (HAS-DET) :DEF :DET-CLASS
         (NO-SONS) (NPTRACE :P :N) (NIL) (NO-CONTRACT) (-CONJ)))
```

Note **(NPTRACE :P :N) (NIL)** as the value of the NP's **TRACE** arguments.

[9]Some researchers who have tried to implement a Montague style compositional semantics using a GPSG syntax have reported that the meta-rule mechanism creates problems for the semantics. Thus, while not having meta-rules may increase the size of the object grammar, this may prevent problems in the area of semantics.

Constituents that may be traces, but are not, specify that the two trace arguments must be the same. For example, all the rules for non-trace NPs have a left hand side of the form:

(2) **(NP ... :TRX :TRX ...)**

Constituents that may contain traces, but which are not traces, use these properties of constituents that can be traces in a fashion that is sometimes described as "trace threading". Here is an example involving a VP with two NPs:

```
(3) ((VP (AGR :P :N) (REALNP :REAL) :MOOD (AUXV (W (W (W (W (W (0AUX))))))
         (NOT-NEG)) (WH-) :TRX :TRZ :CONTRACTX (-CONJ))
    (V :CONTRACT (DITRANSITIVE (TAKES-ACTIVE)) :P :N :MOOD)
    (NP :NSUBCATFRAME :AGRZ (REALNP :REALX) (-POSS :POSSCLASS) (WH-)
        (OBJ (AGR :P :N)) :DET-STATE :DEF :DET-CLASS :SONS
        :TRX :TRY :CONTRACT :CONJC)
    (NP :NSUBCATFRAMEX :AGRW (REALNP :REALY) (-POSS :POSSCLASSX) (WH-)
        :CASEX :DET-STATEX :DEFX :DET-CLASSX :SONSX :TRY :TRZ
        :CONTRACTY :CONJD))
```

The important thing to notice about rule (3) is that VP has the variables **:TRX :TRZ** as the value of its **TRACE** arguments, the first NP has the variables **:TRX :TRY** and the second NP has **:TRY :TRZ**. If an actual VP built by this rule contains an NP trace, the value of its first **TRACE** argument is **(NPTRACE ...)** with appropriate values for its **PERSON** and **NUMBER** arguments and the value of its second **TRACE** argument is **(NIL)**. This, in turn, means that the value of the first **TRACE** argument of the first NP (designated by **:TRX** in the rule) must also be **(NPTRACE ...)** and the second trace argument of the second NP (designated by **:TRZ** in the rule) must be **(NIL)**. In addition, the rule also specifies that the second **TRACE** argument of the first NP and the first trace argument of the second NP (both designated by **:TRY** in the rule) must be the same.

Now, let us observe what happens if the first NP is a trace: its first **TRACE** argument is bound to **(NPTRACE ...)** and the second argument is **(NIL)** (by rule (1)). This value, in turn, is passed along to the first trace argument of the second NP (by the identity of the arguments specified by rule (3)). In turn, the VP rule (3) and the NP rule (2) both require the second **TRACE** argument to also be **(NIL)**. This situation can be graphically represented as follows (omitting all features except for **TRACE**:

```
[(VP ... (NPTRACE ...) (NIL) ...)  (V ...)
                                   (NP ... (NPTRACE ...) (NIL) ...)
                                   (NP ... (NIL) (NIL) ...)]
```

If, instead, the second NP is questioned, the first NP cannot be a trace. Its first **TRACE** argument is bound to **(NPTRACE ...)**. Rule (2) specifies that both of its **TRACE** arguments must be the same, so the value of its second **TRACE** argument is also **(NPTRACE ...)**. Rule (3) requires that the first **TRACE** argument of the second NP be identical to the second **TRACE** argument of the first NP, i.e. also **(NPTRACE ...)**. Rule (3) also requires that the second NP have **(NIL)** as the value of its second **TRACE** argument, which is consistent with rule (1) but with no other rule introducing NPs. This situation can be graphically represented as follows:

```
[(VP ... (NPTRACE ...) (NIL) ...)  (V ...)
                                   (NP ... (NPTRACE ...) (NPTRACE ...) ...)
                                   (NP ... (NPTRACE ...) (NIL) ...)]
```

In general, the ACFG grammar enforces the prohibition against more than one trace in a single clause by allowing only two trace flags on constituents that bear them and by "threading together" the trace flags of constituents that might be traces in the way illustrated. Though this mechanism was illustrated here with only two possible trace constituents on the right hand side of a rule, it can work for arbitrary numbers of such constituents. All that is required is that the traceflags of all the constituents that can be traces (or contain traces) are "threaded together" in the manner illustrated; see the top-level S rule on page 15, for another example of this.

2.1.5 Comparison with other Formalisms

There are currently many annotated context-free grammar formalisms, with associated computational implementations. In addition to Definite Clause Grammars and Heidorn's PNLP formalism, there are also: Lexical Functional Grammar (LFG) [5]; Generalized Phrase Structure Grammar (GPSG) [14]; PATR-II [45]; Head-Driven Phrase Structure Grammar (HPSG) [38]; and Tree Adjoining Grammars (TAGs) [28], [49]. Shieber [46] provides an overview of complex feature based grammars and of some of the ways in which the different formalisms mentioned here differ from one another.

Our experience with the BBN ACFG grammar may prove instructive for the evaluation of the facilities which are absolutely necessary in a complex feature based grammar, since it is a relatively "stripped down" version of such a grammar, like its immediate parent, DCG. While many of the other formalisms mentioned here support the use of one or more relatively sophisticated mechanisms such as feature disjunction, feature negation, metarules, optional arguments, and the use of attribute-value pairs, rather than positional arguments, the BBN ACFG, like DCGs, uses none of these mechanisms. Each grammatical category has a fixed number of obligatory, positional arguments, each of which is limited to a fixed number of values. This typing constraint, which is not found in standard DCGs, and which is the principle difference between the BBN ACFG formalism and DCGs, has been of great usefulness to us in developing a large grammar (currently over 800 rules). Since arguments are obligatory and are restricted in the values they can take on, it is relatively straightforward to have a simple syntactic checker that makes sure that all grammar rules are well-formed. In a grammar as large as the BBN ACFG, having the ability to automatically make sure that all rules are well-formed is no small advantage. We have found no need for most of the advanced facets of other complex feature based grammars, with the possible exception of disjunction, which will probably be added in a restricted form in the future.

2.2 The Parsing Algorithm

The BBN Spoken Language System uses a parsing algorithm which is essentially that of Graham, Harrison, and Ruzzo [15], henceforth, GHR. This algorithm, in turn, is based on the familiar Cocke-Kasami-Younger (CKY) ([19], [29], [54]) algorithm for context-free grammars. The original CKY algorithm could not be used to parse the BBN ACFG since that algorithm requires that a grammar be in Chomsky Normal Form (CNF), i.e. that each rule introducing non-terminal symbols—essentially the parts of speech, as opposed to the terminal symbols (lexical items and grammatical formatives)—be of the form

$A \rightarrow B\ C$

with exactly two non-terminal symbols on the right hand side. A grammar for a natural language will contain rules that deviate from CNF in the following ways:

rules with 0 symbols on the right hand side

the rules that introduce traces, discussed above in Section 2.1.4, are of this type:

```
((NP ...))
```

Such rules are often called *empty rules*.

rules with only 1 symbol on the right hand side

such as the rules introducing intransitive verb phrases, as in:

```
((VP ...)  (V ... (INTRANSITIVE :NPTYPE) ...))
```

Such rules are often called *chain rules*.

rules with more than 2 symbols on the right hand side

such as the rule introducing ditransitive verb phrases, as in:

```
((VP ...)  (V ... (DITRANSITIVE (TAKES-ACTIVE)) ...)  (NP ...)  (NP
...))
```

Nevertheless, the CKY algorithm is quite simple and powerful: it starts with the terminal elements in a sentence and builds successively larger constituents that contain those already found and constructs all possible parses of the input. The GHR algorithm maintains this aspect of the control structure of the CKY algorithm without forcing the grammar to be in CNF. It does this by adding several mechanisms to CKY. All the chain rules of the grammar are collected into a special table that is consulted by the parser to determine if a chain rule is possible at any given point in the parse. All the empty rules of the grammar are collected into a similar table. Finally, for rules with more than two symbols on the right hand side, the mechanism of *dotted rules* is used. A dotted rule is like an ordinary rule, except that the right hand side is divided into two parts by a dot. This dot, in effect, makes the rule look as if it were in CNF. During the course of a parse, the parser will move the dot from the beginning of the right hand side of a rule to its end as the elements of the right hand side are found. Consider the following rule and its dotted rule equivalents:

$A \rightarrow B\ C\ D$
[Constituent A consists of the sub-constituents B C D]
$A \rightarrow .\ B\ C\ D$
[A rule that constructs an A; the parser has not yet found any of its sub-constituents]
$A \rightarrow B\ .\ C\ D$
[A rule that constructs an A; the parser has found a B and is now looking for a C and a D]

A → B C. D
[A rule that constructs an A; the parser has found a B and a C is now looking for a D]
A → B C D.
[A constituent of type A has been found]

The parsing process can be summarized as follows:

1. The input is a sequence of words $w_1, w_2, \ldots w_n$
2. Rules are of the form: $A \rightarrow BCD$
3. Dotted rules are of the form: $A \rightarrow B.CD$
4. Dot movement:

$$\begin{aligned} \text{if } A \rightarrow B.\ C\ D &\stackrel{*}{\Rightarrow} \text{input}[i, j] \\ \&\ C \rightarrow E\ F.\ &\stackrel{*}{\Rightarrow} \text{input}[j, k] \\ \text{then } A \rightarrow B\ C.\ D &\stackrel{*}{\Rightarrow} \text{input}[i, k] \end{aligned}$$

where $A \rightarrow B.CD \stackrel{*}{\Rightarrow} input[i,j]$ means that the symbol B in the dotted rule $A \rightarrow B.CD$ *derives* word sequence $w_i, w_{(i+1)}, \ldots, w_j$ in the input.

5. Partial parses are stored in a chart:

$$\mathbf{dr}[i,k] = \{A \rightarrow B.C \mid B \stackrel{*}{\Rightarrow} input[i,k]\}$$

where $\mathbf{dr}[i,k]$ denotes the set of dotted rules *deriving* the input from i to k.

The GHR algorithm finds all the dotted rules that derive an input sentence; this is another way of saying that it will find all the parses for a sentence. Looking at the parsing algorithm as a way of specifying all the grammatical word sequences of English, we may give the algorithm as in Figure 2-2.

This algorithm contains three nested loops. On the outer level is a loop ranging over *ending* word positions k; at the second level is a loop ranging over *starting* word positions i. Implicitly, at the inner level is a third loop that ranges over intermediate word positions j, $\forall j, i \leq j \leq k$. The parser attempts to combine dotted rules from $dr[i,j]$ (spanning $input[i,j]$) with symbols (completed dotted rules with the dot at the end) from $dr[j,k]$ (spanning $input[j,k]$) to form larger constituents in $dr[i,k]$ (spanning $input[i,k]$).

The procedure used to build constituents out of previously found constituents does not involve simple matching but rather the process of *unification*, which matches the feature values in the different elements of a rule, as specified in the rule. As Section 2.1 showed, features may themselves be complex expressions, so that unification is a recursive process, which has significant implications for computation complexity. Since the GHR algorithm, like the CKY algorithm, deals with context-free grammars, rather than context-free grammars annotated with features, the use of unification is an extension to the GHR algorithm; see [16] and [17] for full details.

One useful result of our work on extending the GHR algorithm to handle annotated context free grammars (ACFGs) is the discovery that there is a class of ACFGs, *depth-bounded ACFGs*, for which the parsing algorithm is guaranteed to find all parses and halt [17]. Depth-bounded ACFGs are characterized by the property that the depth of a parse tree cannot grow unboundedly large unless the length of the string also increases. In effect, such grammars do not permit rules in which a category derives only itself and no other children; such rules do not seem to

For k=1 *to* N

For i=k−1 *to* 0 *by* −1

If i+1 = k

$\mathbf{dr}[i,k]=\{A\rightarrow w.\alpha) \mid w=input[k-1,k]=w_k\}$

Else

$\mathbf{dr}[i,k]=\{(A\rightarrow\alpha B.\beta \mid$

$(A\rightarrow\alpha.B\beta) \in \mathbf{dr}[i,j]$

$\&(B\rightarrow\gamma.) \in \mathbf{dr}[j,k]\}$

$\forall j, i<j<k$

$\cup$

$\{(A\rightarrow B.\beta) \mid (B\rightarrow\alpha.) \in \mathbf{dr}[i,k]\}$

$\forall\ (A\rightarrow B\beta) \in \mathbf{P}$

where

n	is the length of the input in words
w	is a variable ranging over terminal symbols (words)
dr[i,k]	is the set of dotted rules that span the input sentence from the ith through kth positions
input[i,k]	is the portion of the input sentence from the ith through kth positions
P	is the set of grammar rules (productions)

Figure 2-2: BBN ACFG Parsing Algorithm

be needed for the analysis of natural languages, so computational tractability is maintained without sacrificing linguistic coverage. The fact that the parsing algorithm for this class of ACFGs halts is a useful result, since parsers for complex feature based grammars cannot be guaranteed to halt, in the general case. By restricting our grammars to those that satisy depth-boundedness, we can be sure that we can parse input utterances bottom-up and find all parses without the parser going into an infinite loop.

The decision to use an "all paths" parsing algorithm may seem strange, given that other researchers, most notably Marcus [32], [33], [34], have imposed strict determinism as an important constraint on the operation of a natural language processing system. The BBN Spoken Language System uses an algorithm that finds all parses for a number of reasons. The most important of these for the speech understanding task is the fact that speech input is highly ambiguous; the words in the input and their starting and ending positions are all uncertain, as well as their syntactic analysis. Because of this uncertainty, it seems most appropriate that the algorithm used in the integrated speech understanding system should return all possible parses and use all available knowledge sources—acoustical scores, semantics, and pragmatics and discourse information, as they become incorporated into the system—to rank the parses. However, even for text input, a non-deterministic algorithm may be preferable.

An important issue that must be resolved is the interface of deterministic parsing algorithms with a semantic

interpretation component. Typically, deterministic parsers are purely syntactic parsers that do not map the structures obtained into any sort of semantic representation. Since the parse structures obtained by deterministic methods, particularly those in [33] and [34], are sometimes radically different from those normally postulated—for example, the parser may produce a "forest", instead of a tree—there needs to be an independent indication that the structures obtained are in fact useful ones for semantic interpretation and for practical applications. A more serious problem for the strict determinism position is the lack of any detailed mechanism to handle the case where the parser chooses incorrectly and must reprocess an input sentence. Since the parser is deterministic, it is not clear how the parser could ever obtain a different parse for a given utterance, no matter how many times it is processed. Finally, there seem to be genuinely ambiguous sentences (often called cases of "constructional homonymity"— [9], especially p. 28)), such as "Flying airplanes can be dangerous", where it does not seem likely that a single parse can or should be deterministically produced.

For all these reasons, we have felt that it is not appropriate to use a deterministic parsing algorithm. However, while the BBN ACFG parser does not use a deterministic algorithm, the problem that determinism addresses—the potential for huge numbers of parses—still remains and must be faced. We will defer discussion of this issue to Section 2.6.4, which deals with this problem after more details of grammar coverage are introduced.

2.3 The Implementation of the BBN ACFG Formalism

This section describes how the grammar formalism described in Section 2.1 and the parsing algorithm described in Section 2.2 are implemented. The entire system is implemented in Common LISP. There is also an implementation of the parsing algorithm in C that is currently under development.

2.3.1 The Syntactic Type System

As was noted earlier, the BBN ACFG formalism is strongly typed. Syntactic type declarations are made with the function **defrulesyntax**, which takes a list of type declarations as its argument. After the application of **defrulesyntax**, the LISP variable ***rulesyntax*** is bound to a list of all the currently valid grammatical symbols, their arguments, and the permissible values of their arguments. Grammar rules are specified by setting the LISP variable ***inputrules*** to a list of the current grammar rules.

2.3.2 The Lexicon and The Morphology Component

Lexical entries are represented as Common LISP structures. Each lexical entry is associated with a unique "name", which may be different from its base form or "spelling" This disparity between name and spelling arises when there is more than one syntactic category associated with a given base form; for example, *close* as adjective and verb; *ship* as noun and verb; etc. The BBN ACFG lexicon allows each lexical entry to have exactly one syntactic category; hence, when a given "spelling" can be associated with more than one category, it must have a separate entry for each category, and, hence, a separate name for each entry. However, the name and spelling of a lexical entry are typically the same. As an example, we present a "pretty printed" version of the entry for *turn*. (The name of the entry appears immediately to the right of the opening square bracket.)

```
[TURN
    Spelling: turn
    Category: V
    Rules: ((V (NO-CONTRACT) (TRANSPART (ONPARTICLE) :PASSIVE) :P :N
    :MOOD)
            (V (NO-CONTRACT) (TRANSPART (OFFPARTICLE) :PASSIVE) :P :N
    :MOOD))
    Subcat-Features: ((TRANSPART (ONPARTICLE) :PASSIVE)
                      (TRANSPART (OFFPARTICLE) :PASSIVE))
    Semantic-Entries: (TURN-ON
                       TURN-OFF)
    Semantic-Rule: (TURN-ON
                    TURN-OFF)
    Edit-History: Translated by Ingria, 2/12/88 14:25:46
                  added semantics: sboisen 3/24/88]
```

The slots **Spelling**, **Category**, **Subcat-features**, **Semantic-Entries**, and **Edit-History** (as well as **Lex-Features**, **Morph-Features**, **Comments**, and **Examples**, which do not appear in this entry) are all permanently stored in the lexicon. The slots **Rules** and **Semantic-Rule** are filled in when the lexicon is loaded into the system.

Semantic-Rule contains a list of semantic rules that are used by the semantic component. The rules in **Semantic-Rule** are distinguished by two properties: (1) they are in the internal form used by the semantics components, rather than the external form for human consumption; (see Section 3.1.3.7 for more details); and (2) for categories that specify subcategorization information, they are in a one-to-one match-up with the subcategorization features in **Subcat-Features**. The external form of the semantic rule that is stored in the lexicon is the value of the slot **Semantic-Entry** or **Semantic-Entries**. A lexical entry uses **Semantic-Entry** to store its semantics if (1) it is not a category that takes subcategorization frames; or (2) it is a category that takes subcategorization frames, but there is only one semantic rule for all its subcategorization frames. When the lexicon is loaded, the semantic rule in **Semantic-Entry** is converted to the internal semantic form. For lexical entries that are members of categories that do not take subcategorization frames, the translated form of the rule is simply stored in **Semantic-Rule**. For lexical entries that are members of categories that do take subcategorization frames, a list is made that is equal to the length of the **Subcat-Features** slot, with as many copies of the translated semantic rule as are needed to fill out the list; see the lexical entry for **AIRCRAFT-CARRIER** below for

an example of this. A lexical entry uses **Semantic-Entries** to store its semantics if it is a category that takes subcategorization frames, it has more than one subcategorization frame, and there is a separate semantic rule for each subcategorization frame. The value of **Semantic-Entries** is a list of semantic rules; each rule in this list is translated when the lexicon is loaded, and the list of the translated rules is made the value of **Semantic-Rule**. The lexical entry for **TURN** illustrates this.

Rules contains a list of syntactic rules that are filled in as much as possible on the basis of the information in the lexical entry. As has already been stated, each category is declared as having a fixed number of arguments, which appear in a particular order and which can take on a fixed range of values. When a lexical entry is processed, a set of rule "skeleta" is built up on the basis of subcategorization information (stored in **Subcat-Features**) and other syntactic information—e.g. for a noun, whether it is count or mass—(stored in **Lex-Features**), which are used to fill in the various feature values for the associated rule. Features whose values are not specified by information in the lexical entry are filled in in one of two ways. The LISP variable ***default-value-alist*** specifies the default value for various features when they are not filled in by a specific lexical entry. For example, the verb feature **CONTRACT-FLAG**[10] is specified as **(NO-CONTRACT)**. Features that are not filled in by either the individual lexical entry or by the default specification are filled in as variables.

Typically, the features left as variables are morphological features that can only be filled in for a particular inflected form. For example, in the entry for "turn" given above, the person (**:P**), number (**:N**), and mood (**:MOOD**) features are all left variables. Fully specified virtual rules—i.e. rules with constants as the values of the morphological features—are produced by the function **readings**, which takes a word, in the form of a LISP atom, as input and returns a list of the virtual rules compatible with that word as output. For example, the result of applying **readings** to "turns" is:

```
(readings 'turns) =>
(#<SS V TURN-OFF> #<SS V TURN-ON>)
(dolist (thing *) (describe thing)) =>
#<SS V TURN-OFF> is a SS
   SYNTAX:    (V (NO-CONTRACT) (TRANSPART (OFFPARTICLE) 0)
                 (3RD) (SINGULAR) (INDICATIVE (PRESENT)))
   SYN-RULE: ((V (NO-CONTRACT) (TRANSPART (OFFPARTICLE) :PASSIVE)
                 (3RD) (SINGULAR) (INDICATIVE (PRESENT))) (TURNS))
   SEM-RULE: TURN-OFF
   CHILDREN: (NIL)
   SCORE:    0
#<SS V TURN-OFF> is implemented as an ART-Q type array.
It uses %ARRAY-DISPATCH-WORD; it is 6 elements long.
#<SS V TURN-ON> is a SS
   SYNTAX:    (V (NO-CONTRACT) (TRANSPART (ONPARTICLE) 0)
                 (3RD) (SINGULAR) (INDICATIVE (PRESENT)))
   SYN-RULE: ((V (NO-CONTRACT) (TRANSPART (ONPARTICLE) :PASSIVE)
                 (3RD) (SINGULAR) (INDICATIVE (PRESENT))) (TURNS))
   SEM-RULE: TURN-ON
   CHILDREN: (NIL)
```

[10]The function of this feature is discussed in Section 2.5.5.

```
   SCORE:    0
#<SS V TURN-ON> is implemented as an ART-Q type array.
It uses %ARRAY-DISPATCH-WORD; it is 6 elements long.
```

Note that the data structures produced by **readings**, which, in turn, are used by the parser, are Common LISP structures called SSs (for Syntax-Semantics). These structures have 5 slots:

syntax A modified version of the syntactic rule; this is the form actually used by the parser.

syn-rule The original form of the syntactic rule used to build the current constituent; this is the form stored in the grammar (in the case of a grammar rule) or produced by **readings** (in the case of a virtual rule).

sem-rule The semantic rule associated with the current syntactic rule.

children A list of the children, if any, of the current constituent. The children are also SSs.

score A numerical value; this is used by the "score parser", described in Section 2.6.4.

Once the virtual rules associated with a particular inflected form have been computed, they may be stored in a hash table, indexed by the lexical item. Caching prevents the system from incurring the additional computational expense of recomputing the same virtual rules every time the same form is encountered. However, during system development work, when lexical entries may be changed frequently, caching may be a nuisance, since it can prevent changed rules from being computed by the system. Therefore, caching is controlled by the LISP variable ***cache-readings***: when its value is **NIL**, caching is disabled; when is value is non-**NIL**, caching is enabled.

So far, we have considered only the way in which lexical items that consist of a single word are treated by the system. Collocations—multi-word lexical items—are handled in a slightly different manner. They are stored the same way, as the following entry for "aircraft carrier" shows. The only new slot here is **Head**, which contains an integer, indicating, in a 0-based ordering scheme, which element of the collocation bears inflection; in this case, its value is 1, indicating that "carrier" is the inflected part of the collocation "aircraft carrier".

```
[AIRCRAFT~CARRIER
    Spelling:           aircraft carrier
    Head:               1
    Category:           N
    Morph-Features:     (PARADIGM S)
    Rules:              ((N (TAKESPREMODIFIER) (AGR (3RD) :N) (REALNP
                            (NONUNITNP (-PRO (MISCNP)))) (+DET
                            (COUNT-N (BASIC-COUNT))) (-CONJ))
                         (N (INTRANSNOUN) (AGR (3RD) :N) (REALNP
                            (NONUNITNP (-PRO (MISCNP)))) (+DET
                            (COUNT-N (BASIC-COUNT))) (-CONJ)))
    Subcat-Features:    ((TAKESPREMODIFIER)
                         (INTRANSNOUN))
    Semantic-Entry:     CARRIERS
    Semantic-Rule:      (CARRIERS
                         CARRIERS)
    Comments:           copied semantics from carrier; added for irus
                        queries --- rjpi
    Edit-History:       Translated by Ingria, 2/12/88 14:25:46]
```

The virtual rules associated with collocations, however, are not computed by the **readings** function. Rather, they are computed when the lexicon is loaded, for both the base form and all inflected forms of the collocation, and are added to the table of grammar rules used by the system. That is, the virtual rules associated with collocations are

treated as if they were ordinary grammar rules. This is moderately useful in the text parser, since it allows the parser to operate without a separate mechanism that checks the input for collocations. Moreover, this treatment of collocations is necessary for the lattice parser.

As a utility, the macro **pw** exists to print the lexical information associated with a given lexical item or grammatical formative. **pw** is given the spelling of a non-collocational lexical item, as a LISP atom, and returns *all* the entries associated with that spelling:

```
(pw close) =>

The spelling ``CLOSE'' is associated with 2 lexical entries.
Here they are:

[CLOSE-V
    Spelling:            close
    Category:            V
    Rules:               ((V (NO-CONTRACT) (TRANSITIVE (REALNP :REAL)
                             :PASSIVE) :P :N :MOOD))
    Subcat-Features:     ((TRANSITIVE (REALNP :REAL) :PASSIVE))
    Semantic-Entry:      CLOSE-V
    Semantic-Rule:       (CLOSE-V)
    Examples:            ("When was the CASREP closed?")
    Edit-History:        Added for Parlance IDB corpus
                         ---ingria, 9/21/88]
```

```
[CLOSE
    Spelling:              close
    Category:              ADJ
    Rules:                 ((ADJ (INTRANSADJ) :AGR :DEGREE (PRENOMADJ)
                                 (PREDICATIVE-ADJ) (NIL) (NIL) (-CONJ))
                           (ADJ (CASE-COMP (TOCASE)) :AGR :DEGREE
                                 (PRENOMADJ) (PREDICATIVE-ADJ) (NIL)
                                 (NIL) (-CONJ))
                           (ADJ (INTRANSADJ) :AGR :DEGREE (NOT-PRENOMADJ
                                 (PREDADJ)) (PREDICATIVE-ADJ) (NIL) (NIL)
                                 (-CONJ))
                           (ADJ (CASE-COMP (TOCASE)) :AGR :DEGREE
                                 (NOT-PRENOMADJ (PREDADJ))
                                 (PREDICATIVE-ADJ) (NIL) (NIL) (-CONJ)))
    Subcat-Features:       ((INTRANSADJ)
                            (CASE-COMP (TOCASE)))
    Semantic-Entries:      ((LAMBDA #:X (GROUPS THINGS) (APPLY SMALLER-THAN
                               (TUPLE ((APPLY EFL-DISTANCE-BETWEEN (TUPLE
                               (#:X (STIPULATE (GROUPS THINGS) #:*QUA*))))
                               #:SVAR))))
                            (LAMBDA (X Y) ((GROUPS THINGS) (GROUPS THINGS))
                               (SMALLER-THAN (EFL-DISTANCE-BETWEEN X Y) SVAR)))
    Semantic-Rule:         ((LAMBDA #:X (GROUPS THINGS) (APPLY SMALLER-THAN
                               (TUPLE ((APPLY EFL-DISTANCE-BETWEEN (TUPLE
                               (#:X (STIPULATE (GROUPS THINGS) #:*QUA*))))
                               #:SVAR))))
                            (LAMBDA #:T6520 (TUPLES ((GROUPS THINGS)
                               (GROUPS THINGS))) (APPLY SMALLER-THAN
                               (TUPLE ((APPLY EFL-DISTANCE-BETWEEN (TUPLE
                               ((ELT 1 #:T6520) (ELT 2 #:T6520)))) #:SVAR)))))
    Edit-History:          Translated by Ingria, 2/12/88;
                           modified by Scha, 6/27/88
                           Remerged---9/22/88 18:31:38, Ingria]
```

A collocation is passed to **pw** as a list:

```
(pw (bering strait)) =>

[BERING~STRAIT
    Spelling:            Bering Strait
    Head:                1
    Category:            N
    Lex-Features:        (DETFLAG :DET
                          COUNTCLASS (PROPER-N (MISC-PROPER)))
    Rules:               ((N (TAKESPREADJECTIVE) (AGR (3RD) :N)
                             (REALNP (NONUNITNP (-PRO (MISCNP))))
                             (+DET (PROPER-N (MISC-PROPER))) (-CONJ))
                          (N (INTRANSNOUN) (AGR (3RD) :N) (REALNP
                             (NONUNITNP (-PRO (MISCNP)))) (+DET
                             (PROPER-N (MISC-PROPER))) (-CONJ))
                          (N (TAKESPREADJECTIVE) (AGR (3RD) :N)
                             (REALNP (NONUNITNP (-PRO (MISCNP))))
                             (-DET (PROPER-N (MISC-PROPER))) (-CONJ))
                          (N (INTRANSNOUN) (AGR (3RD) :N) (REALNP
                             (NONUNITNP (-PRO (MISCNP)))) (-DET
                             (PROPER-N (MISC-PROPER))) (-CONJ)))
    Subcat-Features:     ((TAKESPREADJECTIVE)
                          (INTRANSNOUN))
    Semantic-Entry:      BERING-STRAIT
    Semantic-Rule:       (BERING-STRAIT
                          BERING-STRAIT)
    Comments:            a water place --- dgs
    Edit-History:        Translated by Ingria, 2/12/88 14:25:46]
```

pw also finds the grammar rules that introduce grammatical formatives:

```
(pw who) =>

``WHO'' is introduced directly by the grammar in the rules:

(NP (INTRANSNOUN) (AGR (3RD) (SINGULAR)) (REALNP (NONUNITNP (+PRO (PRO-NP
    (FORBIDS-ANTECEDENT))))) (-POSS (NOT-POSS)) (WH+ (QWH) (NPTRACE (3RD)
    (SINGULAR))) :CASEX (HAS-DET) (+DEF) (MISC-DET) (NO-SONS) (NIL) (NIL)
    :CONTRACT (-CONJ)) ->
    (WHO)
(LAMBDA #:P PREDS (LAMBDA #:X AGENTS (APPLY #:P #:X)))

(NP (INTRANSNOUN) (AGR :P :N) (REALNP (NONUNITNP (+PRO (PRO-NP (FORBIDS-
    ANTECEDENT))))) (-POSS (NOT-POSS)) (WH+ (RELWH) (NPTRACE :P :N))
    :CASEX (HAS-DET) (+DEF) (MISC-DET) (NO-SONS) (NIL) (NIL) :CONTRACT (-
    CONJ)) ->
    (WHO)
(LAMBDA #:P PREDS (LAMBDA #:X AGENTS (APPLY #:P #:X)))
```

2.3.3 The Parser

Our parsing algorithm is implemented by the **parse** function. Before actually entering the main computational loop, **parse** calls the **pre-process** function on its input. This is a very simple function, which splits off contractions, negatives, and possessives into separate terminal elements. Currently, it also discards punctuation, since we have not yet been able to determine a satisfactory way of making use of punctuation. Some examples of its application are:

```
Isn't Frederick in Port? => IS |N'T| FREDERICK IN PORT
Vinson won't downgrade to C2. => VINSON WILL |N'T| DOWNGRADE TO C2
Kennedy's mission readiness is M1. => KENNEDY |'S| MISSION READINESS IS M1
Eisenhower cannot arrive before March 2 => EISENHOWER CAN NOT ARRIVE
                                           BEFORE MARCH 2
```

Internally, **parse** uses SS structures that represent grammar rules or the virtual rules produced by **readings**. It returns three values: a list of all the SS structures that represent a parse that spans the entire input utterance (including **NIL**, the null list, when the parser has found no parse); the original input utterance; and the form of the input utterance produced by **pre-process**.

For assigning pronominal reference and for semantic processing, each SS is transformed into a data structure called a CFG-NODE.[11] This is implemented as a flavor object in the LISP Machine implementation of the system and as a structure in the general Common Lisp implementation. CFG-NODEs have the following slots:

category The category of the current constituent.

semantics The semantic rule associated with the current constituent; corresponds to the **sem-rule** slot of an SS.

syntax The syntactic rule associated with the current constituent; corresponds to the **syn-rule** slot of an SS.

parents A list containing the parent, if any, of the current constituent. The parent is also a CFG-NODE.

children A list containing the children, if any, of the current constituent. Each child is also a CFG-NODE.

possible-antecedents
Used for pronominal reference; see Section 2.4 for details.

impossible-antecedents
Used for pronominal reference; see Section 2.4 for details.

preterminal-p **T** or **NIL**, depending on whether the constituent is a pre-terminal node or not.

terminal-p **T** or **NIL**, depending on whether the CFG-NODE is associated wtih a terminal element (word or grammatical formative) or not.

terminal-el If the CFG-NODE is associated with a terminal element, the terminal element is stored here; **NIL** otherwise.

CFG-NODEs are also used by the intra-sentential pronominal reference mechanism, described in the next section.

[11] CFG-NODEs are also used for display purposes in the LISP Machine implementation.

2.4 The Pronominal Reference Mechanism

The BBN Spoken Language System currently has syntactic mechanisms for assigning antecedents to pronouns within sentences. The algorithm used for assigning antecedents is inspired by the indexing scheme of Chomsky [10], augmented by tables analogous to the "Table of Coreference" of [25].

In Section 2.4.1 the empirical and theoretical background to the intra-sentential reference mechanism is sketched out. In Section 2.4.2, the actual intra-sentential algorithm used is described in detail.

2.4.1 Theoretical Background

While most computational systems are interested in the potential antecedents of pronouns, work in generative grammar by Lasnik [31] and Reinhart [42] has led to the conclusion that sentential syntax is responsible for assigning possible antecedents to bound anaphors (reflexives, such as "himself", "herself", "themselves", etc., and the reciprocals "each other" and "one another") but not to personal pronouns ("he", "she", "they", etc). In the case of personal pronouns, sentential syntax only determines the syntactically *impossible* antecedents. This latter procedure is called *disjoint reference*, since the impossible antecedents can not even overlap in reference with the pronoun; compare the cases in sentences (4) and (5), where the underlined items are non-identical in reference, with those in (6) and (7), where they are non-overlapping in reference. In (4) and (5), "he" and "him" cannot refer to "John" (non-identical reference); while in (6) and (7) "John" cannot be a member of the set referred to by "they" and "them" (non-overlapping or disjoint reference).

(4) He likes John.
(5) John likes him.

(6) They like John.
(7) John likes them.

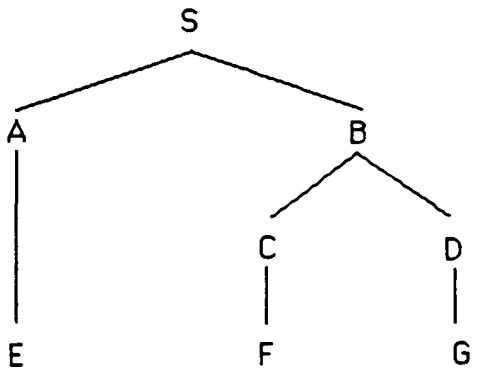
Disjoint reference is even more noticeable with first and second person pronouns: here, disjoint reference does not merely produce impossible interpretations, it actually produces ungrammaticality:

(8) *I like me.
(9) *I like us.
(10) *We like me.
(11) *You like you.

A crucial notion both for assigining antecedents to bound anaphors and for establishing disjoint reference between Noun Phrases is that of *c-command*, a structural relation. Briefly, as defined by Reinhart [42], node A c-commands node B iff the branching node most immediately dominating A also dominates B, and A does not dominate B. Figure 2-3 illustrates this.[12]

Essentially, the relation between c-command and reference phenomena is the following:

[12]The bracketed statements will be discussed later.



A c-commands B, C, F, D, and G [E c-commands B, C, F, D, and G]
B c-commands A and E
C c-commands D and G [F c-commands D and G]
D c-commands C and F [G c-commands C and F]

Figure 2-3: C-Command

1. A non-pronominal NP cannot overlap in reference with any NP that c-commands it.
2. The antecedent of a bound anaphor must c-command it.[13]
3. A personal pronoun cannot overlap in reference with an NP that c-commands it.[13]

Condition 1 is motivated by sentences such as those in (12), where the underlined pronouns "he", "him", "they", and "them" must be disjoint in reference with "John". In each case, the pronouns c-command the NP "John". In (12a) "he"/"they" is in the subject position, and so c-commands "John", in the direct object slot. In (12b) the pronouns ("He", "They") are once again in the subject position, and "John" is the object of a preposition, itself contained in the direct object of the sentence. Finally, in (12c), the NP "John" appears as the object of a preposition, which is c-commanded by the subject ("He", "They") and the direct object ("him", "them").

(12) a. He likes John — They like John
b. He likes pictures of John — They like pictures of John
c. He told them about John — They told him about John

Condition 2 is motivated by examples such as those in (13), where the reflexive pronoun "himself" and its antecedent(s) are bracketed. As in the corresponding examples in (12), "himself" either appears as a direct object (13a), the object of a preposition within the direct object (13b), or as a prepositional object (13c). In all cases, the c-commanding subject ("John") is a possible antecedent; in (13c), where the c-commanding object NP "Bill" is added, it is also a possible antecedent.

(13) a. [John] likes [himself]
b. [John] likes pictures of [himself]
c. [John] told [Bill] about [himself]

Condition 3 is motivated by examples such as those in (14), where underlining again indicates disjoint reference. The pronoun under consideration ("him" or "them") always appears as an object or prepositional object

[13] Within a minimal syntactic domain; this will be explained shortly.

and is disjoint in reference to the c-commanding subject "John" (in (14a,b,c)) and to the c-commanding direct object "Bill" in (14c).

(14) a. John likes him — John likes them
b. John likes pictures of him — John likes pictures of them
c. John told Bill about him — John told Bill about them

While condition 1 is unconditionally true, conditions 2 and 3 are subject to a further constraint, which we might term minimality. When a possessive appears in the determiner position of an NP, thereby c-commanding the other constituents of the NP, it blocks Noun Phrases external to the NP from being antecedents of a bound anaphor within the NP; it also blocks external NPs from participating in disjoint reference with personal pronouns within the NP (and so allows them to be overlapping in reference). This is illustrated in (15)--(17) (underlining indicates disjoint reference; bracketing indicates co-reference). The subject NP in (16) is not a possible antecedent for the reflexive; while the subject NP in (17) need not be disjoint in reference with the underlined pronoun. Compare (16) with (13b) and (17) with (14b); the presence of the possessive makes the NP dominating the reflexive or personal pronoun a minimal domain and so makes NPs external to this minimal NP inaccessible.

(15) He likes Bill's pictures of John — They like Bill's pictures of John
(16) John likes [Bill's] pictures of [himself]
(17) [John] likes Bill's pictures of [him] — [John] likes Bill's pictures of [them]

In the Government-Binding framework of Chomsky [11], these generalizations are captured by the Binding Theory—a set of well-formedness conditions on syntactic structural representations annotated with subscript and superscript "indices". The paradigm assumed there is Generate and Test: indices are freely assigned and the Binding Conditions are applied to rule in or rule out a particular assignment. Clearly, from a computational standpoint this is grossly inefficient. However, in earlier work, Chomsky ([10], pp. 38--44) proposed an indexing mechanism that captures these facts in a procedural manner.

His proposal assigns each non-bound anaphor (i.e. non-pronominal NP or personal pronoun) the pair (r,A) where r (for *R*eferential index) is a non-negative integer and A (for *A*naphoric index) is a set of such integers. In the first pass, r and A are assigned from left-to-right in a depth-first manner. Each non-bound anaphor NP is assigned a unique r; in addition, the r index of each NP c-commanding it is added to its A index. This set of integers indicates all the other NPs with which it is disjoint in reference. For non-pronominal NP's, only one pass is needed; (18) illustrates this:

(18) $John_2$ told $Bill_{(3,\{2\})}$ about $Fred_{(4,\{2,3\})}$

The indices here indicate that "John", "Bill", and "Fred" are all disjoint in reference

In the case of personal pronouns, a second pass is necessary. Consider example (17), repeated here as (19), after the first pass:

(19) $John_2$ likes $Bill's_{(3,\{2\})}$ pictures of $him_{(4,\{2,3\})}$

The indexing at this stage indicates that "Bill" is disjoint in reference from "John" and that "him" is disjoint in reference from "Bill", which is correct, and also from "John", which is not. To correct this Chomsky has a second pass, in which the r indices of certain NPs are removed from the A index, thereby allowing them to serve as

potential antecedents.[14] After this second pass, the indexing is as follows:

(20) John$_2$ likes Bill's$_{(3,\{2\})}$ pictures of him$_{(4,\{3\})}$

At this stage "John" is no longer specified as being disjoint in reference with "him".

While this procedure is an improvement on [11] it still has some computational problems. First, it requires (at least) two passes, the first to assign r and A indices, the second to modify some of those assignments by removing r indices from the already assigned A indices. Second, the potential antecedents of a personal pronoun are only implicitly represented: any NP whose r index is not a member of that pronoun's A index set is a syntactically permissible antecedent, but the set of permissible antecedents is not enumerated; for example, in (16), "John" is indicated as a potential antecedent of "him" by virtue of the fact that its r index, 2, is not part of the A index of "him", and in no other way.

In the next section, we present the implemented mechanism for establishing disjoint reference and coreference relations that is inspired by Chomsky's proposal but which differs from it regarding these two features. First, it is single pass: while there may not be a great computational loss in the two-pass character of Chomsky's original proposal, clearly it is cleaner to do things in one pass. Moreover, the mechanism is extensionally richer than Chomsky's in that it also handles cases of backwards-pronominalization and split-antecedence; it manages to do more and still be single pass.

Second, it explicitly indicates the potential antecedents of a personal pronoun. Again, this is more desirable than leaving this information implicit; besides the potential (and perhaps small) computational savings of not needing to recompute this information, there is the more general consideration that we are not interested in creating syntactic representations for their own sakes, but to make use of them in actual systems. Explicitly representing antecedence information for personal pronouns can contribute to this goal. Finally, the mechanism is general enough that it can be extended to other phenomena that involve c-command: we have already used it to implement a treatment of "$\overline{\text{N}}$ anaphora".

The algorithm also crucially uses a version of c-command that is modified from Reinhart's.[15] Under this definition, a node c-commands its sisters and any nodes dominated by its sisters. The difference between the two definitions is illustrated in Figure 2-3: the bracketed statements are true under Reinhart's definition of c-command, because no branching category intervenes between the c-commanding and c-commanded nodes, but not under that used in the implemented algorithm, since there is no sisterhood among the nodes. We have found this modified definition to be easier to implement; moreover, various researchers have pointed out problems with Reinhart's definition that the modified definition solves. Since Reinhart's version makes crucial use of the notion of branching, rather than sisterhood, this predicts certain asymmetries that do not seem to exist. For example, an intransitive verb, which is in a non-branching VP, will c-command the subject NP, while a transitive verb, which is in a branching VP, will not. However, intransitive sentences do not show the effects on the subject position that would follow if

[14]For details of this second pass, see Chomsky ([10], pp. 38--44).

[15]In fact, it is equivalent to the *in construction with* relation of Klima [30] p. 297), which inspired c-command.

the subject NP actually were c-commanded by the verb.[16]

2.4.2 The Algorithm

The algorithm applies as a post-process to a completed parse tree. Each node in the tree is a CFG-NODE data structure; two of its slots are used for establishing pronominal reference:

:possible-antecedents

a list of all the nodes that can be co-referent or overlapping in reference with the node.

:impossible-antecedents

a list of all the nodes that are disjoint in reference with it.

The algorithm is essentially a left-to-right, depth first traversal of the entire parse tree; it also makes a "blackboard" type use of two tables: ***table-of-proforms*** and ***table-of-antecedents***; these tables are inspired by the "Table of Coreference" of Jackendoff [25]. Finally, the algorithm uses the notion "current cyclic node", which is the S or NP that most immediately dominates the node being processed, and the related notions of "internal" and "external" nodes. Internal nodes are dominated by the current cyclic node; external nodes c-command or dominate the current cyclic node.

The algorithm uses two major procedures: **pass-down-c-commanding-nodes** and **update-node**. **pass-down-c-commanding-nodes** is responsible for actually traversing each node in the tree; whenever it reaches a node, it applies the procedure **update-node** to that node with three arguments: the current cyclic node, a list of external c-commanding nodes, and a list of internal c-commanding nodes. **update-node**, in turn, performs the correct pronominal assignment. The algorithm used by **update-node** is shown in Figure 2-4 in a LISP-type notation.

when *node* is not pronominal [I]
  add each node in **EXTERNAL-NODE-LIST** to *node*'s **IMPOSSIBLE-ANTECEDENTS** slot, when that node is an NP [I.A]
  add each node in **INTERNAL-NODE-LIST** to *node*'s **IMPOSSIBLE-ANTECEDENTS** slot, when that node is an NP [I.B]
  add *node* to the **POSSIBLE-ANTECEDENTS** slot of each node in ***TABLE-OF-PROFORMS***, when that node is not already in *node*'s **IMPOSSIBLE-ANTECEDENTS** slot [I.C]
  add *node* to ***TABLE-OF-ANTECEDENTS*** [I.D]

Clause [I], repeated here in a more English-like notation for expository purposes, implements condition 1 (non-pronominal NPs). Since there are no minimality conditions on disjoint reference for non-pronominal NPs, all NP nodes c-commanding a non-pronominal NP are added to its **:impossible-antecedents** slot ([I.A] and [I.B]). This handles sentences such as those in (12) and (15). While it might seem odd to specify that a non-pronominal NP has no antecedents, this information is useful in handling cases of backwards pronominalization, as in (21).

(21) [His] mother loves [John]

[16] See [2] for more details.

```
(defun update-node (node current-cyclic-node external-node-list internal-node-list)
    (case (category node)
      (NP
        (cond ((non-pronominal node) [I]
               (loop for other-node in external-node-list [I.A]
                    do (when (equal (category other-node) 'NP)
                             (add other-node (impossible-antecedents node))))
               (loop for other-node in internal-node-list [I.B]
                    do (when (equal (category other-node) 'NP)
                             (add other-node (impossible-antecedents node))))
               (loop for pro in *table-of-proforms* [I.C]
                    do (when (not (member pro (impossible-antecedents node)))
                             (add node (possible-antecedents pro))))
               (push node *table-of-antecedents*)) [I.D]
              ((bound-anaphor node) [II]
               (case current-cyclic-node
                 (S [II.A]
                   (loop for other-node in internal-node-list
                        do (when (equal (category other-node) 'NP)
                                 (add other-node (possible-antecedents node)))))
                 (NP
                   (if (some #'(lambda (x)
                                 (equal (category x) 'NP))
                               internal-node-list)
                       (loop for other-node in internal-node-list [II.B]
                            do (when (equal (category other-node) 'NP)
                                     (add other-node (possible-antecedents node))))
                       (loop for other-node in external-node-list [II.C]
                            do (when (equal (category other-node) 'NP)
                                     (add other-node (possible-antecedents node)))))))))
              ((pronoun node) [III]
               (case current-cyclic-node
                 (S [III.A]
                   (loop for other-node in internal-node-list
                        do (when (equal (category other-node) 'NP)
                                 (add other-node (impossible-antecedents node))))
                   (loop for other-node in external-node-list
                        do (when (equal (category other-node) 'NP)
                                 (add other-node (possible-antecedents node)))))
                 (NP
                   (if (some #'(lambda (x)
                                 (equal (category x) 'NP))
                               internal-node-list)
                       (block minimal-NP [III.B]
                              (loop for other-node in internal-node-list
                                   do (when (equal (category other-node) 'NP)
                                            (add other-node (impossible-antecedents
                                                             node))))
                              (loop for other-node in external-node-list
                                   do (when (equal (category other-node) 'NP)
                                            (add other-node (possible-antecedents
                                                             node)))))
                       (loop for other-node in external-node-list [III.C]
                            do (when (equal (category other-node) 'NP)
                                     (add other-node (impossible-antecedents node)))))))
               (loop for NP in *table-of-antecedents* [III.D]
                    do (when (not (member NP (impossible-antecedents node)))
                             (add NP (possible-antecedents node))))
               (push node *table-of-proforms*)))))) [III.E]
```

Figure 2-4: The Reference Algorithm

Clause [I.C] handles backwards pronominalization by making use of information in ***table-of-proforms***, a table of all the pronouns encountered so far in the course of the tree walk.[17] After **update-node** has added all c-commanding NP nodes to the **:impossible-antecedents** slot of a non-pronominal NP, it then searches ***table-of-proforms*** for any pronouns that are not on its **:impossible-antecedents** list· whenever it finds one, it adds the current non-pronominal NP to the pronoun's **:possible-antecedents** list. The last thing **update-node** does in processing a non-pronominal NP is to add it to ***table-of-antecedents*** ([I.D]), whose use will be explained shortly.

when *node* is a bound anaphor [II]
  when *current-cyclic-node* is S [II.A]
  add each node in **INTERNAL-NODE-LIST** to *node*'s **POSSIBLE-ANTECEDENTS** slot, when that node is an NP

  when *current-cyclic-node* is NP

  if there is an NP in *internal-node-list*
  add each node in **INTERNAL-NODE-LIST** to *node*'s **POSSIBLE-ANTECEDENTS** slot, when that node is an NP [II.B]
  else add each node in **EXTERNAL-NODE-LIST** to *node*'s **POSSIBLE-ANTECEDENTS** slot, when that node is an NP [II.C]

Clause [II] implements condition 2 (bound anaphors). Clause [II.A] handles bound anaphors when they appear as children of S, as in (13a) and (13c). Clause [II.B] handles cases where a bound anaphor appears inside an NP with a possessive, which induces minimality, as in (16). Finally, clause [II.C] handles cases where a bound anaphor appears inside an NP and there are no minimality effects, as in (13b).

when *node* is a pronoun [III]
  {
  when *current-cyclic-node* is S [III.A]

  add each node in **INTERNAL-NODE-LIST** to *node*'s **IMPOSSIBLE-ANTECEDENTS** slot, when that node is an NP
  add each node in **EXTERNAL-NODE-LIST** to *node*'s **POSSIBLE-ANTECEDENTS** slot, when that node is an NP

  when *current-cyclic-node* is NP
  if there is an NP in *internal-node-list*
  add each node in **INTERNAL-NODE-LIST** to *node*'s **IMPOSSIBLE-ANTECEDENTS** slot, when that node is an NP
  add each node in **EXTERNAL-NODE-LIST** to *node*'s **POSSIBLE-ANTECEDENTS** slot, when that node is an NP **[III.B]**
  else
  add each node in **EXTERNAL-NODE-LIST** to *node*'s **IMPOSSIBLE-ANTECEDENTS** slot, when that node is an NP **[III.C]**
  }
  add each node in ***TABLE-OF-ANTECEDENTS*** to *node*'s **POSSIBLE-ANTECEDENTS** slot, when that node is not already in *node*'s **IMPOSSIBLE-ANTECEDENTS** slot [III.D]
  add *node* to ***TABLE-OF-PROFORMS*** [III.E]

Clause [III] implements condition 3 (personal pronouns). Since personal pronouns are subject to minimality

[17]This table is filled in by Clause [III.E].

restrictions, like bound anaphors, the general shape of [III] is similar to that of [II]. However, since both disjoint reference and potential reference information is important for the interpretation of personal pronouns, there are several additional subclauses. Clause [III.A] handles personal pronouns when they appear as children of S, as in (14a) and (14c). It also handles examples such as

(22) [John] regrets the fact that Bill dislikes [him]

where "Bill", which is internal to *him*'s current cyclic node (S), is disjoint in reference with *him*, while "John", which is external, is potentially coreferent.

Clause [III.B] handles cases where a personal pronoun appears inside an NP with a possessive, thereby inducing minimality, as in (17). Clause [III.C] handles cases where a pronoun appears inside an NP and there are no minimality effects, as in (14b). Clause [III.D], handles sentences like (23)

(23) [John's] mother loves [him]

in which a non-pronominal NP that does not c-command a pronoun serves as its antecedent. As was noted above, each non-pronominal NP is added to the ***table-of-antecedents***. When **update-node** has added all the appropriate c-commanding nodes to the **:impossible-antecedents** slot of a personal pronoun, it then adds any NPs on ***table-of-antecedents*** that are not already on the pronoun's **:impossible-antecedents** slot to its **:possible-antecedents** slot. Finally, when **update-node** is finished processing a pronominal NP node, it adds it to ***table-of-proforms** ([III.E]).

2.4.3 Extensions of This Treatment to Other Phenomena

As Figure 2-4 shows, it is possible to extend the algorithm to handle other phenomena that involve c-command by modifying the top level **CASE** statement to dispatch on other categories. In fact, the same function is used to handle examples of "N-bar anaphora"; i.e. cases where the head N is either "one" (which has been argued to be an anaphor for N-Bar's, i.e. an N and its complements, but not a full NP) or phonologically null (∅), which seem to have the same possibilities for antecedents.

(24) Give me a list of ships which are in the gulf of Alaska that have casualty reports dated earlier than Esteem's oldest one.

(25) Is the Willamette's last problem rated worse than Wichita's ∅?

```
(N-BAR
  (when (pro-n-bar-p cfg-node)
        (loop for other-node in external-node-list
              do (when (and (equal (category other-node) 'NP)
                            (possible-pro-n-bar-antecedent other-node)
                            (add (get-son-of-category other-node 'N-BAR)
                                 (possible-antecedents node))))))))
```

or, in more English-like notation:

in the case when *node* is an N-BAR
if *node* is a pro N-BAR
for each node in **EXTERNAL-NODE-LIST**, when that node is a permissible antecedent for *node*, add the N-BAR of that node to *node*'s **POSSIBLE-ANTECEDENTS** slot

This clause is considerably simpler that those that handle disjoint reference and co-reference phenomena for noun phrases: only external nodes are involved and only forward antecedence is necessary.

2.4.4 Extensions to the Algorithm

Currently, the algorithm does not do any checking on the potential antecedents of a pronoun or bound anaphora to see if they agree in person and number.[18] For bound anaphors, this is straightforward: a bound anaphor and its antecedent must agree in person and number. For personal pronouns, on the other hand, the situation is more complicated and may be seen in the following table:

Person	Number	Antecedent
(1ST)	(SINGULAR)	None allowed
(2ND)	(SINGULAR)	None allowed
(3RD)	(SINGULAR)	Must be (3RD) (SINGULAR)
(1ST)	(PLURAL)	No restrictions
(2ND)	(PLURAL)	No restrictions
(3RD)	(PLURAL)	Must be (3RD)

The restriction that first person (''I'', ''me'') and second person (''you'') singular pronouns do not allow antecedents reflects the fact that they are like proper nouns: the first person singular pronoun refers to the speaker and the second person singular pronoun refers to the listener, although these roles are contextually determined. Third person pronouns, on the other hand, can take an antecedent: in the singular (''he'', ''him'', ''she'', ''her'', ''it'') this antecedent must also be in the singular. In the plural (''they'', ''them'') , however, the only restriction on a potential antecedent is that it must be third person. This loosening of the restrictions on potential antecedents in the plural is motivated by cases of '' split antecedents'', such as

(26) John told Bill that they should leave.

in which more than one NP antecedes a third person pronoun: here ''John'' and ''Bill'', together, antecede ''they''.[19]

First person (''we'', ''us'') and second person (''you'') pronouns also allow split antecedents:

[18]Currently, NPs in the BBN ACFG are not specified for gender, so this cannot be checked.

[19]Note that not requiring the antecedent of a third person plural pronoun to agree with it in number will allow ''someone'' to be the antecedent of ''their'' in examples such as ''Someone lost their watch''. Since this usage is becoming common, this is probably a feature of this treatment.

(27) a. I told John that we should go.
b. I told you that we should go.
c. Bill told you that you should go.
d. [I] told you that [you] should go.
e. [John] told [Bill] that [we] should go.
f. [John] told [Bill] that [you] should go.

Note that a first person plural pronoun allows split antecedents only if at least one of them is itself first person; contrast (27a) and (27b) with (27e). Similarly, a second person plural pronoun allows split antecedents only if at least one of them is also second person—contrast (27c) with (27f)—but not if one is first person; contrast (27c) with (27d).[20] Note that the agreement check sketched out here does not enforce this requirement; a separate post-process—either syntactic or semantic—would be necessary to see that it is met.

Currently, there is one case of pronominal reference within NPs that is not handled by the algorithm. As the desriptions of the portions of the algorithm that handle bound anaphors and personal pronouns show, there is a great deal of parallelism in the treatment of these items. In fact, it is usually said that if the antecedent-anaphor relation holds between two positions, disjoint reference also holds between them; see examples (13) and (14), and (16) and (17). However, there is one position where this generalization does not hold: the possessive position of an NP. A bound anaphor is possible here, but a pronoun in the same position is not subject to disjoint reference. See (28), where underlining indicates co-reference in both examples.

(28) a. [The men] read [each other's] books
b. [The men] read [their] books

(28a) is correctly handled by the algorithm as already outlined; there is no internal c-commanding NP for "each other's", so the algorithm will assign the NP "the men" to its `:possible-antecedents` slot. However, the algorithm as stated will also assign the NP "the men" to the `:impossible-antecedents` slot of "*their*" in (28b), for a similar reason. Hence, the algorithm must be modified to make an exception for a possessive pronoun whose cyclic node is NP, and assign external c-commanding NP nodes to its `:possible-antecedents` slot.

Finally, the current algorithm basically handles single clause utterances. There are certain phenomena that arise in multi-clause utterances that it does not handle.

[20]The generalization seems to be that if at least one member of a split antecedent is first person, the personal pronoun must be first person; if at least one member of a split antecedent is second person, the personal pronoun must be second person; and if no member of a split antecedent is first or second person, the personal pronoun is third person. This seems to mirror the facts regarding person agreement on verbs for conjoined NPs (discussed below in Section 2.5.6.4). If at least one conjunct in a conjoined NP is first person, verb agreement must be first person; if at least one conjunct in a conjoined NP is second person, verb agreement must be second person; and if no conjunct in a conjoined NP is first or second person, verb agreement is third person.

2.5 Qualitative Measures of Coverage

This section describes the coverage of the current ACFG grammar in descriptive linguistc terms. Quantitative measures of coverage are presented in Section 2.6.

2.5.1 The Top Level Constructions

Like a standard context free grammar, the BBN ACFG has a designated initial symbol, the function letter **START**, which takes no arguments. **START** currently introduces the following types of constituents.

The BBN ACFG grammar currently handles the following types of utterances:

- Declarative sentences ("The Eisenhower is in the Indian Ocean.")
- Interrogative sentences ("Is the Eisenhower in the Indian Ocean?") in various types; the full spectrum is discussed in Section 2.5.2.1.
- Imperatives[21] ("Display the Indian Ocean.")
- NP utterances ("The ships in the Indian Ocean")
- Utterances made up of single interjections—single words or fixed phrases that constitute complete utterances ("Over and out.", "Roger.")
- Utterances made up of an interjection followed by a declarative clause ("Yes, the Eisenhower is in the Indian Ocean.")
- Utterances made up of an interjection followed by an interrogative clause ("No, is the Eisenhower in the Indian Ocean?")
- Utterances made up of an interjection followed by an imperative ("Yes, display the Indian Ocean.")

2.5.2 Clausal Constructions

The current grammar handles clauses that comprise full utterances (so-called "matrix clauses") as well as subordinate clauses of different types.

2.5.2.1 Matrix Clauses

The following types of matrix clauses are currently handled:

- Declarative clauses ("The Eisenhower is in the Indian Ocean.")
- Yes-no questions ("Is the Eisenhower in the Indian Ocean?") and Content ("WH") questions ("Who is in the Indian Ocean?")

Content questions may involve various types of constituents:

[21] These are treated as VPs in the current grammar, rather than S(entence)s.

- Noun Phrases such as "who", "what", "how many ships", etc.
- Adjective Phrases such as "how long", "how much longer", etc.
- Locative and temporal expressions such as "where", "when", etc.
- Adverbial expressions such as "how", "why", etc.

Both yes-no and content questions involve a process of "subject-aux inversion" in which an "auxiliary element" and the subject are transposed; the ACFG grammar handles all such cases:

- modals ("Must Eisenhower go to the Indian Ocean?")
- perfective "have" ("Has Eisenhower gone to the Indian Ocean?")
- progressive "be" ("Is Eisenhower going to the Indian Ocean?")
- passive "be" ("Is Eisenhower deployed to the Indian Ocean?")
- "main verb" "be" ("Is Eisenhower in the Indian Ocean?")
- auxiliary "do" ("Does Eisenhower have harpoon?")

The negated counterpart of each type is also handled:

- "Mustn't Eisenhower go to the Indian Ocean?"
- "Hasn't Eisenhower gone to the Indian Ocean?"
- "Isn't Eisenhower going to the Indian Ocean?"
- "Isn't Eisenhower deployed to the Indian Ocean?"
- "Isn't Eisenhower in the Indian Ocean?"
- "Doesn't Eisenhower have harpoon?"

Each of these question rules also allows ADVPs (ADVerb Phrases) of a specified type to appear after the subject NP:

- "Has Frederick ever gone to C3 on personnel readiness?"
- "When was Eisenhower last in the Indian Ocean?"
- "How soon will Wasp next chop to Atlantic Fleet from PACFLT?"

2.5.2.2 Subordinate Clause Constructions

Subordinate clauses (that is, clauses that make up a subpart of a complete utterance) can be divided into two types:

Complement clauses
: These are clauses that are introduced as complements to lexical categories, such as verb, noun, or adjective, and which are permitted or forbidden by individual lexical items. For example, verbs like "believe" and "say" take complement clauses introduced by "that" while "go" and "come" do not.

Adjunct clauses
: These are clauses that are introduced in specified structural positions of phrases and clauses, independent of the exact lexical item that heads the phrase or clause. For example, noun phrases allow relative clauses introduced by WH words or "that" but do not permit clausal adjuncts introduced with "because"; compare "A/The man who was old came in." with "*A/The man because he was old came in."

Complement Clauses

The following types of complement clauses are currently handled:

- Finite clauses introduced by "that":
  1. Complement clauses that permit an optional "that" ("We believe that Eisenhower is in the Indian Ocean." and "We believe Eisenhower is in the Indian Ocean.")
  2. Complement clauses that require "that" ("I order that Eisenhower sail to the Indian Ocean." but not "*I order Eisenhower sail to the Indian Ocean.")

These two types of complement clauses are often called indirect statements.

- Complement clauses that require "if" ("We wonder if Eisenhower is in the Indian Ocean.")
- Complement clauses that require "whether" ("We wonder whether Eisenhower is in the Indian Ocean.")
- Complement clauses that require a WH phrase ("We wonder who/which ship is in the Indian Ocean.")

These three types of complement clauses are often called indirect questions.

Adjunct Clauses

The following types of adjunct clauses are currently handled:

- Relative clauses introduced by a WH word ("Get the maximum speeds for carriers which are in Astoria.") or "that" ("List carriers that are C5 on equipment.")

  The current grammar also handles "extraposed" relative clauses with a WH word or "that" ("Five ships arrived that were C3.")

  The grammar also handles "stacked" relative clauses (i.e. multiple relative clause on the same head noun) ("Are there any ships which are located in China Sea that are C2.") and conjoined relative clauses, even if one has "that" and the other has a WH word ("Give me a list of the carriers that are M3 on ASW and which are in New York.") In both these cases, where both "that" and a WH word appear, they may appear in either order.

  The grammar also handles so-called "free" or "headless" relative clauses, such as "Whoever is C1 should go to the Indian Ocean." Free relatives with "where", such as "Get me a list of their maximum sustained speeds and where they are enroute to." are also treated.

- Clausal adjuncts that appear at the end of finite clauses ("Eisenhower is in the Indian Ocean because there is an emergency there.") or infinitival complements ("We believe Eisenhower to be in the Indian Ocean because it is needed there.")

2.5.2.3 Clausal Conjunction

In addition to conjunction and stacking of relative clauses, there is general clausal conjunction with "and" and "or".

2.5.2.4 The Immediate Constituents of Clauses

Clauses have as their immediate constituents (i.e. the elements that make up a clause) the subject noun phrase, a verb phrase (i.e. the verb and its complements) and various adjuncts. Currently, the following elements can appear as adjuncts:

- A participial clause ("Redraw the area updating the display.")
- A participial clause introduced by a preposition ("Redraw the area without updating the display.")
- Adverbial expressions of various types ("now", "yesterday", "on the twenty second of May")
- Locative and temporal expressions ("at seventy degrees north twenty three degrees east", "in the Indian Ocean")
- Prepositional phrases introduced by "with" that contain a noun phrase and a predicate phrase ("with areas off", "with system switches set to default values", "with the Shasta's in bright green")
- Extraposed relative clauses with "that" or a WH consitutent (see Subsection 2.5.2.2).

- Sentential adjuncts introduced by various complementizers ("after", "because", "unless", "until", etc).
- Various prepositional phrases: "with" ("Redraw the chart of Formosa with redefined overlays."), "without" ("Show me Rathburne's track without the overlay."), "for" ("Turn echo off for tracks whose lat-lon is hooked."), "instead of" ("What if the Dubuque's propulsion type was diesel instead of gas turbine?")[22]

2.5.3 Verb Phrase Constructions

As was explained in Section 2.1.1.1, lexical items are linked to the subcategorization frame(s) in which they can appear by means of a subcategorization feature whose values are associated with separate complement rules. For VPs, this is done by means of the argument **SUBCATFRAME** on Vs. There are currently 49 verbal subcategorization frames, which are derived from an extensive survey [23] of the literature concerning verb subcategorization in English (e.g. [1], [6], [41], [50], [51], [52], [53]). Since each of these features may be associated with more than one VP rule (e.g. the feature for transitive verbs is associated with both an active and a passive VP) there are more than 49 separate VP subcategorization rules; in fact, there are 83 such rules. Discusssion of these rules is beyond the range of this section but see [24] for a detailed discussion of each.

2.5.4 Auxiliary Constructions

The BBN ACFG provides the full range of standard VP auxiliary phenomena in English. Since there are no meta-rules in the formalism (see Section 2.1.3) there is a separate rule for each type of auxiliary element:

- modals ("Eisenhower must go to the Indian Ocean.")
- perfective "have" ("Eisenhower has gone to the Indian Ocean.")
- progressive "be" ("Eisenhower is going to the Indian Ocean.")
- passive "be" ("Eisenhower is deployed to the Indian Ocean.")
- "main verb" "be" ("Eisenhower is in the Indian Ocean.")
- auxiliary "do" ("Eisenhower does have harpoon.")

There is a fixed order to auxiliary elements in English:

- modal elements—which include *can*, *could*, *may*, *might*, *must*, *shall*, *should*, *will*, *would*, as well as *'ll* as a contraction for *will*—precede
- perfective *have*, which precedes
- progressive *be*, which precedes
- passive *be*, which precedes
- main verbs

In addition, "main verb" *be* follows progressive *be* and is in complementary distribution with passive *be* and lexical main verbs.

[22]Prepositional phrases as a general category are not freely allowed in sentential adjunct position, but are either specified individually or as members of some more restricted class, such as locative/temporal phrases. See Section 2.5.9 for further discussion.

The following sentence shows this order:

You must have been being bad or they wouldn't have gotten angry at you.

While this order is fixed, not all auxiliary elements need to appear all the time; as long as they appear in the correct order, any subset of them can appear:

Eisenhower must leave.
Eisenhower must have left.
Eisenhower must have been leaving.
Eisenhower has left.
Eisenhower is leaving.
Eisenhower must be leaving.
...

In addition, "dummy" *do* can only appear if no other auxiliary element appears. It appears in questions, in negated sentences, and in emphatic statements:

Did Eisenhower reach the Indian Ocean?
Eisenhower didn't reach the Indian Ocean.
Eisenhower did reach the Indian Ocean!

English also contains restrictions on the placement of the sentence level negative elements *not* and its contracted form *n't*. These negative formatives appear following the first auxiliary element in a clause. If there is no other auxiliary element in the sentence, *do* appears, since negation cannot appear with a simple main verb in English:

*John went not/wentn't.
*John not/Johnn't went.
vs.
John didn't go.

The ACFG grammar handles both the order and optionality behavior of the English auxiliary system. It also handles the placement of negation.

To capture both the ordering relation among auxiliary elements and the fact that any subset of them can appear, auxiliary elements are treated as the first constituent in a VP followed by a VP which, in turn, can begin with any auxiliary or verbal element that can appear to the right of that auxiliary. The correct ordering is ensured by assigning the VP headed by a particular type of auxiliary to an "auxiliary level" and requiring that the VP which follows that auxiliary be at least of the next highest level. Schematically, this mechanism involves rules of the form:

$$\textbf{VP [AUX-LEVEL}_N\textbf{]} \rightarrow \textbf{AUXILIARY}_N \textbf{ VP [AUX-LEVEL}_{\geq N+1}\textbf{]}$$

This auxiliary level scheme is implemented by a combinator-like treatment of "auxiliary level". Each VP has an **AUX** argument that takes **AUXV** as one of its arguments. **AUXV**, in turn, takes **AUXLEVEL** as its first argument. **AUXLEVEL** has the following arguments:

```
:AUX
:AUXX
:AUXY
(0AUX)
(W AUXLEVEL)
```

(0AUX) may be viewed as the zero element of a combinator system while **W** may be viewed as succesor. Given these primitives, it is easy to impose the requirements that the VP following an auxiliary element be of the right "level". Given the rule:

```
((VP :AGR :NPTYPE :MOOD (AUXV (W (0AUX)) :NEG) (WH-) :TRX :TRY (-CONJ))
 (MODAL)
 (VP :AGR :NPTYPE (BASE) (AUXV (W (W :AUX)) (NOT-NEG)) (WH-) :TRX :TRY
    :CONJC))
```

the modal may be followed directly by:

- perfective *have* (introduced by a VP with **AUXLEVEL (W (W (0AUX)))**)
- progressive *be* (introduced by a VP with **AUXLEVEL (W (W (W (0AUX))))**)
- passive *be* (introduced by a VP with **AUXLEVEL (W (W (W (W (0AUX)))))**)
- main verb *be* (introduced by a VP with **AUXLEVEL (W (W (W (W (0AUX))))))**)
- a regular verb (introduced by a VP with **AUXLEVEL (W (W (W (W (W (0AUX)))))))**)

Each of the rules introducing these auxiliary elements in turn requires the VP that they introduce to be at least of the next higher **AUXLEVEL**.

The rule introducing dummy *do* is:

```
((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (0AUX)) :NEG) (WH-) :TRX :TRY (-
    CONJ))
 (V (DO) :P :N :MOOD)
 (VP :AGR :NPTYPE (BASE) (AUXV (W (W (W (W (W (0AUX)))))) (NOT-NEG)) (WH-)
    :TRX :TRY :CONJC))
```

This allows only a VP headed by a main verb to follow *do*, which is correct.

In addition to the **AUXLEVEL** argument, **AUXV** also takes a second argument, **NEGFLAG**. **NEGFLAG** controls the placement of the sentence level negative elements *not* and its contracted form *n't*. In English, these negative formatives appear following the first auxiliary element in a clause. If there is no other auxiliary element in the sentence, *do* appears, since negation cannot appear with a simple main verb in English:

*John went not/wentn't.
*John not/Johnn't went.
vs.
John didn't go.

NEGFLAG is used to enforce this restriction. It has the arguments:

```
(NEG-POSS)
(NOT-NEG)
:NEG
:NEGX
:NEGY
```

Each VP rule introducing an auxiliary element has a counterpart in which a negative element (**(NEG)**) follows the auxiliary. Each VP rule introducing **(NEG)** requires that its **NEGFLAG** be **(NEG-POSS)**. The S rules that introduce VP set the **NEGFLAG** to **(NEG-POSS)**; every other rule that introduces a VP, including the auxiliary VP rules, sets **NEGFLAG** to **(NOT-NEG)**. For example, here is the rule introducing negated perfective **have**:

```
((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (0AUX))) (NEG-POSS)) (WH-) :TRX
    :TRY (-CONJ))
 (V (HAVE) :P :N :MOOD)
 (NEG)
 (VP (AGR :P :N) :NPTYPE (EDPARTICIPLE) (AUXV (W (W (W :AUX)))) (NOT-NEG))
    (WH-) :TRX :TRY :CONJC))
```

Note **(NEG-POSS)** as the value of **NEGFLAG** in the VP introducing *have*. Recall that there are two ways in which a VP introducing *have* can be built: (1) as a direct descendent of S; (2) as a direct descendent of a VP that introduces a modal. Since the rule introducing modals specifies that the following VP must be **(NOT-NEG)** (see the rule on page 48 above), the rule introducing negated *have* is inapplicable. This means that *have* cannot be negated when it follows a modal, whether or not that modal is itself negated. The same is true of all the other rules introducing negated auxiliaries: negation is possible if the auxiliary is the first auxiliary in a clause and impossible otherwise.

2.5.5 Verb Contraction

As is well known (see, e.g. [4]), it is impossible for verbs in English to reduce when they are followed by an "extraction site", e.g. the position from which a WH-element has been dislocated, which is analyzed as being occupied by a trace. Compare:

(28) a. I wonder who the guilty party is
b. *I wonder who the guilty party's

Comparative constructions also produce the same effect and so are also often treated as involving a trace:

(29) a. He is smarter than you are.
b. *He is smarter than you're.

This constraint not only rules out certain utterances, it also rules out certain interpretations of acceptable utterances. For example, consider the next sentence, which can only have the (a) reading and not the (b) reading.

(30) Frederick's speed is greater than Eisenhower's
a. Frederick's speed is greater than Eisenhower's (speed)
b. Frederick's speed is greater than Eisenhower (is)

(where () enclose the elided element, which is the parse found)

Clearly, then, not implementing this constraint allows in ungrammatical word sequences and also produces unwanted parses for grammatical utterances. To avoid these problems, we have implemented a mechanism that prevents verbs from contracting when they are immediately followed by a trace. The mechanism involves the following features:

- V and Modal, the categories that actually contract, are assigned the argument **contract-flag** whose values are: **(can-contract)**, **(no-contract)**, and **:contract**. Contracted versions of verbs and modals have the value **(can-contract)**; uncontracted forms have the value **(no-contract)**.
- NP, ADJP, PP, LOC-TEMP-PHRASE, the categories that can appear next to one of the contracting categories, are also assigned the argument **contract-flag**. The trace version of each one of these categories has the value **(no-contract)**. Non-trace versions have the value **:contract**.
- NEG, the category that introduces the formatives "not" and "n't" also is assigned the argument **contract-flag**. The version of NEG that introduces "n't" has the value **(no-contract)**. The version introducing "not" has the value **:contract**.
- VP is also assigned the argument **contract-flag**, since it introduces the categories NP, ADJP, PP, LOC-TEMP-PHRASE when they are in predicate position. Predicate VPs agree with their NP, ADJP, PP, or LOC-TEMP-PHRASE daughter in the value of **contract-flag**; other VPs make its value a free variable.

The following rules illustrate these features of the mechanism:

```
((V (CAN-CONTRACT) (BE) (3RD) (SINGULAR) (INDICATIVE (PRESENT)))
 (|'S|))
((V (NO-CONTRACT) (BE) (3RD) (SINGULAR) (INDICATIVE (PRESENT)))
 (is))
;; NP trace rule
((NP :NSUBCATFRAME (AGR :P :N) (REALNP :REAL) (-POSS (NOT-POSS)) (WH-)
     :CASEX (HAS-DET) :DEF :DET-CLASS (NO-SONS) (NPTRACE :P :N) (NIL)
     (NO-CONTRACT) (-CONJ)))
;; ordinary count NP rule
((NP :NSUBCATFRAME (AGR :P :N) (REALNP (NONUNITNP :NON-UNIT))
     (-POSS (NOT-POSS)) :WH :CASEX (HAS-DET) :DEF :DET-CLASS
     :SONS :TRX :TRX :CONTRACT (-CONJ))
 (DETERMINER :N (COUNT-DET :DEF :DET-CLASS) (NONNULLNBAR) :WH
    (-PARTITIVE))
 (OPTNONPOSADJP (AGR :P :N) (NON-POSITIVE :NON-POS) (PRENOMADJ))
 (OPTADJP (AGR :P :N) (PRENOMADJ))
 (N-BAR :NSUBCATFRAME (AGR :P :N) (REALNP (NONUNITNP :NON-UNIT))
        (+DET (COUNT-N :C-SOURCE)) (NIL) (NIL) :CONJ)
 (OPTNPADJUNCT (AGR :P :N) :SONS))
;; ADJP trace rule
((ADJP :ADJSUBCATFRAME :AGR :DEGREE :POSIT (WH-)
        (ADJPTRACE :ADJSUBCATFRAME :AGR :POSIT) (NIL)
        (NO-CONTRACT) (-CONJ)))
;; ordinary ADJP rule
((ADJP :ADJSUBCATFRAME :AGR :DEGREE :POSIT :WH :TRX :TRX :CONTRACT
    (-CONJ))
 (DEGREESPEC :WH :DEGREE)
 (ADJ-BAR :ADJSUBCATFRAME :AGR :DEGREE :POSIT :WH :TRX :TRX))
;; Rule introducing predicate VP
((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (W (W (0AUX)))))) :NEG)
     (WH-) :TRX :TRY :CONTRACTX (-CONJ))
```

```
 (V :CONTRACT (BE) :P :N :MOOD)
 (VP (AGR :P :N) :NPTYPE (EDPARTICIPLE) (PRED) (WH-) :TRX :TRY
    :CONTRACT :CONJC))
;; predicate nominal: ''(John is) a doctor''
((VP (AGR :P :N) (REALNP :REAL) :MOOD (PRED) (WH-) :TRX :TRY
     :CONTRACT (-CONJ))
 (NP :NSUBCATFRAME (AGR :PERSONX :NUMX) (REALNP :REALX)
     (-POSS :POSSCLASS) (WH-) (OBJ (AGR :P :N)) :DET-STATE
     :DEF (NON-QUANT :NQ) :SONS :TRX :TRY :CONTRACT :CONJB))
;; predicate adjective: ''(John is) smart''
((VP (AGR :P :N) (REALNP :REAL) :MOOD (PRED) (WH-) :TRX :TRY
     :CONTRACT (-CONJ))
 (ADJP :ADJSUBCATFRAME (AGR :P :N) :DEGREEX (NOT-PRENOMADJ (PREDADJ))
     (WH-) :TRX :TRY :CONTRACT :CONJA))
```

The foregoing rules show the mechanism that is necessary to prevent contraction from over-applying. The main verb "be" must agree in its **contract-flag** argument with a following predicate VP. Predicate VPs, in turn, agree in this feature with their predicate element. When the predicate VP contains a predicate element that is a trace, the value of **contract-flag** is **(NO-CONTRACT)**, which prevents the appearance of "'s", which is **(CAN-CONTRACT)**. On the other hand, when the predicate element is not a trace, it sets its **contract-flag** argument to a variable, allowing both "'s" and "be" to appear.

Since our system has a prepass mechanism that separates out contracted elements as separate terminals, it is also necessary to prevent sequences such as "'s n't" and "'ll n't". The next set of rules handles this:

```
((MODAL (CAN-CONTRACT))
 (|'LL|))
((MODAL (NO-CONTRACT))
 (shall))
((MODAL (NO-CONTRACT))
 (will))
((NEG :CONTRACT)
 (not))
((NEG (NO-CONTRACT))
 (|N'T|))
;; negated version of predicate VP rule
((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (W (W (0AUX))))) (NEG-POSS))
     (WH-) :TRX :TRY :CONTRACTX (-CONJ))
 (V :CONTRACT (BE) :P :N :MOOD)
 (NEG :CONTRACT)
 (VP (AGR :P :N) :NPTYPE (EDPARTICIPLE) (PRED) (WH-) :TRX :TRY
     :CONTRACTY :CONJC))
;; rule introducing modals and negation
((VP :AGR :NPTYPE (INDICATIVE :TENSE) (AUXV (W (0AUX)) (NEG-POSS)) (WH-)
     :TRX :TRY :CONTRACTX (-CONJ))
 (MODAL :CONTRACT)
 (NEG :CONTRACT)
 (VP :AGR :NPTYPE (BASE) (AUXV (W (W :AUX)) (NOT-NEG)) (WH-) :TRX :TRY
     :CONTRACTY :CONJC))
```

Auxiliary verbs, such as "be", and modals, such as "shall" and "will", must agree in their

contract-flag argument with a following NEG. When the NEG is itself contracted to "n't", the value of its **contract-flag** is **(NO-CONTRACT)**, which prevents the appearance of contracted verb and modal forms, such as "'s" and "'ll", which are **(CAN-CONTRACT)**. On the other hand, when the NEG element is not contracted ("not"), it sets its **contract-flag** argument to a variable, allowing both "'s" and "be", "'ll" and "will", etc. to appear.

The interesting fact about this treatment of verb contraction is the similarity in the way that a trace and an overt contracted element such as "n't" both block contraction of a preceding verb or modal. In the case of "n't", the reason seems fairly clear: English does not seem to allow sequences of contracted (or *cliticized*) elements. The fact that traces also block contraction in a similar way suggests that they may also need to cliticize to a preceding element; since English prohibits a clitic element from attaching to an element which has itself cliticized, the inability of verbs to contract before a trace follows. When the verb contracts, the following trace cannot cliticize; under the hypothesis that traces must cliticize, this will result in ungrammaticality. This observation, in turn, might explain why traces are known to block contraction in other circumstances (e.g. of infinitival "to"; see [26] and [39] for details). For example, "John is the man who I want to succeed." is ambigous between readings in which the object position of "want" and the object position of "succeed" are treated as the extraction site for "who". "John is the man who I wanna succeed." on the other hand, only has the interpretation in which "who" is extracted from the object position of "succeed". Again, if we postulate that traces are cliticizing elements and that cliticization to a reduced element is prohibited in English, this follows: the trace must cliticize to the preceding verb and so blocks contraction of the following "to"—the converse of the situation that we have been considering so far in which the contraction of a preceding auxiliary blocks the contraction of a following trace, resulting in ungrammaticality. While this observation is suggestive, its full ramifications are beyond the scope of this document.

2.5.6 Noun Phrases

Ordinary (i.e. "common" or non-proper) noun phrases consist of the following elements (where parentheses indicate optionality):

(Determiner) (Comparative/Superlative-Adjectives) (Positive-Adjectives) N-Bar (Adjuncts)/(Relative-clauses)

Such common nouns include count nouns, mass nouns, and abstract nouns.

Determiners include those elements typically called determiners: the definite and indefinite articles (*the*, *a*, *an*), demonstratives (*this*, *that*), quantifiers (*all*, *some*, *each*, ...), etc. Determiner structure is discussed in more detail in Section 2.5.6.1.

Comparative and Superlative Adjectives
are introduced into noun phrases separately from positive adjectives for both syntactic and semantic reasons. The semantic argument is simple: it is difficult or impossible to write a semantic rule that will work for an arbitrary number of adjectives that will work for both positive and superlative adjectives. There are several syntactic facts that also point to non-positive adjectives being introduced separately from positive adjectives:

- Superlative and comparative adjectives must precede non-positive adjectives and cannot be intermixed with one another:

the fastest green ships
*the green fastest ships

- Superlative and comparative adjectives preferentially select particular noun phrase adjuncts. Comparative adjectives can co-occur with phrases introduced by *than*:

  a faster ship than the Frederick

  Superlative adjectives can co-occur with phrases introduced by *of*:

  the fastest ship of the unit

  Positive adjectives impose no such restrictions.

- Superlative adjectives require definite determiners, while positive adjectives impose no such restrictions.

Positive adjectives are the base (non-compared) forms of adjectives (''green'', ''fast'').

An unlimited number of adjectives (positive, comparative, or superlative) are introduced via the optionality mechanism mentioned in Section 2.1.2.

N-BAR introduces the head noun of the noun phrase together with its complements, if any. N-BAR rules are discussed in some more detail in Section 2.5.6.3.

Adjuncts (introduced under the node **OPTNPADJUNCT**)

Currently, in an attempt to prohibit an explosion of ambiguity, there is no recursive structure to Noun Phrase adjuncts: unlike S adjuncts, which are recursive. However, multiple adjuncts are allowed, by special rules. Currently, the following constituents can appear as adjuncts:

- Adjective Phrases: These are Adjective Phrases either headed by adjectives that lexically specify a postnominal position (such as *left* in ''the fuel left'') or adjective phrases containing a complement, since adjective phrases containing complements cannot appear prenominally in English; compare ''sufficient fuel'', ''fuel sufficient to get there'', ''*sufficient to get there fuel''. For more discussion of adjective placement, see Section 2.5.7.
- Participial VPs: ''the ship approaching port''
- Passive VPs: ''the ship deployed to the Indian Ocean''
- Prepositional Phrases introduced by *with*, *for*, *from*, or *of*:[23] ''all ships with harpoon'', ''vessels with maximum speeds above seven knots'', ''their arrival for the meeting'', ''problems from the last nine months'', ''equipment of the highest quality''.
- Locative and temporal expressions: ''the meeting on the seventeenth of July''.

Currently, the following sequences of constituents can appear as adjuncts:

- A locative and a temporal phrase: ''any vessel in Everett after fifteenth September''.
- A locative and a date expression: ''all of WESTPAC's carriers in their home ports the seventh of February''.
- A prepositional phrase or locative/temporal phrase and a passive VP: ''all data for the Meteor updated since twenty three hundred hours''.
- A locative/temporal phrase followed by any of the prepositional phrases permissible as NP adjuncts: ''the four ships in the Formosa Strait with the largest fuel capacities''

[23]Prepositional phrases as a general category are not freely allowed in NP adjunct position, but are either specified individually or as members of some more restricted class, such as locative/temporal phrases. See Section 2.5.9 for further discussion.

Relative clauses

Relative clauses with relative *that* (''all ships that are C1'') and WH words (''all ships which are C1''), including ''whose'' (''commanders whose ships are C1'') are handled.

Conjoined relative clauses may also appear here. *That* relatives may be conjoined with WH relatives. ''Stacked'' relative clauses, such as ''ships that are in the Indian Ocean that are C1'', may also appear.

Relative clauses are currently introduced separately from NP adjuncts and may not co-occur with them. This is a historical accident, based on the fact that relative clauses were added to NP rules before other adjuncts were. In the future, relative clauses will be introduced under **OPTNPADJUNCT** and will be allowed to co-occur with other NP adjuncts.

A separate NP structure is distinguished for ''unit NPs'':

NP → (Unit-Det) N-BAR

Unit NPs are head by nouns that represent units of measure, such as ''feet'', ''gallons'', ''days''. They are distinguished from other noun and noun phrase types on both syntactic and semantic grounds:

- Their external distribution is different from that of ordinary common nouns. They can appear as premodifiers to certain positive and comparative adjectives and various types of prepositional phrases: ''three miles long'', ''four weeks old'', ''five pounds heavier'', ''two inches from it'', ''within a foot of him'', etc.
- Their internal distribution is also different from that of ordinary common nouns. They take only a subset of the determiners that common nouns take, do not freely appear with adjectives, and do not seem to allow adjuncts at all; ''one foot long'', ''*every foot long'', ''*some foot/feet long'', ''how many feet long'', ''???three dusty miles long'', ''*five miles in the Indian Ocean long''. They also introduce partitive type expressions, unlike common nouns:[24] ''thirty gallons of fuel'', ''twenty feet of rope'', etc.
- They are given a different semantic treatment from common nouns; see Section 3.2.2.7 for details.

Finally, proper NPs have a still different structure:

NP → (Propdet) N

PROPDET is a category that introduces the article ''the'' which is lexically specified as appearing before certain proper nouns either obligatorily (as in ''the Hague'', ''the Soviet Union'', etc.) or optionally (as in ''(the) Vinson'').

Proper NPs have a flatter structure than ordinary NPs, since they are restricted in the elements that can appear: no determiners, except for ''the'', which is lexically specified; no adjuncts; and only a few adjectives. For the purposes of the Resource Management domain, certain proper nouns—essentially those dealing with land masses or bodies of water, such as ''Atlantic Ocean'' or ''California''—are lexically specified as taking a preceding adjective; e.g. ''northern Atlantic Ocean'', ''southern California'', etc. Such adjectives appear under an N node that also dominates the proper noun itself; this allows the combination of adjective and proper noun to be part of a noun noun modification structure, such as ''northern Atlantic Ocean ships''.

[24]This construction is not handled by the current grammar.

There is also a small grammar of "titles" for proper nouns that is meant to handle not only elements normally considered as titles, such as "Mr.", "Colonel", etc. but also things such as "U.S.S.", which is treated as structurally analogous. To ensure that the right type of title goes with the right type of proper noun, proper nouns are subdivided into different types, based on the title that is possible. Thus, proper nouns of type **(FIRST-NAME)** and **(LAST-NAME)** appear with titles specified as **(PERSONAL-TITLE)**; proper nouns of type **(SHIP-NAME)** appear with titles specified as **(SHIP-TITLE)**; and proper nouns of type **(MISC-PROPER)** do not appear with titles, at all. These distinctions might seem to be "semantic", but since they do have syntactic, distributional ramifications, it seemed appropriate to assume that they have been grammaticalized.

2.5.6.1 Noun Phrase Determiners

The determiner structure of English noun phrases is one of the most complicated areas of English syntax. There are a great number of dependencies between the determiner and the classes of nouns that appear as the head of NP. The determiner itself may have a complex structure with similar dependencies among its elements.

The ACFG grammar ensures that "count nouns" (i.e. nouns that can appear with articles, that can be pluralized, etc.) appear only with count determiners, such as "every" ("every man" vs. "*much man"), etc.; and that non-count nouns (both "mass" and "abstract" nouns) (i.e. nouns that typically do not appear with articles, do not easily pluralize, etc.), appear only with non-count determiners such as "much" ("much sand" vs. "*every sand), etc.

The ACFG grammar allows the full range of determiners, including quantificational determiners, such as "every" and "how many", numerical determiners, including ordinals, cardinals, and fractions, such as "three", "twenty one", "three quarters", as well as the "pre-determiner" elements that appear with them, such as "more than" ("more than six ships", "more than three quarters of the ships") or "all" ("all six ships"), the traditional articles "the", "a", and "an", the demonstratives "this", "that", "these", and "those", the possessive determiners, which include both "possessive pronouns" ("my", "your", "their", etc.) and full scale possessive NPs, as well as other determiner elements, such as "both", "either", "neither", etc. Other, specialized determiner constructions are also handled, such as "the/my last three vacations".

Some determiners can appear without any following noun, such as "those" (as in "All the ships in Diego Garcia are C4 but those in Honolulu are C1.") and such constructions are also handled.

The ACFG grammar also handles "question" determiners, such as "which" and "how many" and partitive determiners, such as "some" and "each" (e.g. "some of the ships", "each of the ships").

To capture the full range of these possibilities, the **DETERMINER** category has two expansions. **DETERMINER** always contains a **DETHEAD**, which introduces the elements traditionally labelled as determiners. **DETHEAD** can also be preceded by **PREDET**, which dominates elements often called predeterminers.

Feature agreement is used to enforce dependencies among the elements of **DETERMINER** as well as between **DETERMINER** and the head NP. Here is the **DETERMINER** functor, with all its arguments:

```
(DETERMINER NUMBER COUNTFLAG NBARFLAG WH PARTFLAG)
```

DETHEAD takes the same arguments, in the same order. **PREDET** takes a single argument, **COUNTFLAG**. The **DETERMINER** and the **DETHEAD** that it contains, as well as the **PREDET**, if any, agree in their features.

NUMBER is the "grammatical number" of the NP and its determiner, which, in turn, is determined by the number of the head N of the NP. Possible values are **(SINGULAR)**, **(PLURAL)**, and the variables **:N**, **:NUMX**, **:NUM**, **:NUMY**, and **:NUMZ**.

COUNTFLAG is the most crucial argument for enforcing correct co-occurrence relations within the determiner itself and between the determiner and the head noun of the NP in which it appears. Here are all the possible arguments to **COUNTFLAG**:

```
(COUNT-DET DEFFLAG DET-CLASS)
(NON-COUNT-DET DEFFLAG DET-CLASS)
:COUNTFLAG
```

COUNT-DET appears on determiner elements that can appear with "count nouns" (i.e. nouns that can appear with articles, that can be pluralized, etc.), such as "every" ("every man" vs. "*every sand"), etc.

NON-COUNT-DET is used for determiners that can only be used with non-count nouns (both "mass" and "abstract" nouns) (i.e. nouns that typically do not appear with articles, do not easily pluralize, etc.), such as "much" ("much sand" vs. "*much man"), etc.

DEFFLAG specifies the definiteness of the determiner element. Possible values are:

- **(+DEF)** for definite determiner elements, such as "the", "this", "that", etc.
- **(-DEF)** for indefinite determiner elements, such as "a", "some", "any", etc.
- **:DEF** the variable value for this feature.

DET-CLASS specifies the "class" of the determiner element. This feature is used to enforce co-occurrence restrictions between the **PREDET** and the **DETHEAD**. It has the values:

(MISC-DET)
appears on **DETHEAD**s that do not appear with **PREDET**s, such as "both", "either", "neither", "enough", etc.

(QUANT) appears on "quantificational" **DETHEAD**s, such as "every" and "how many", that do not appear with **PREDET**s. This feature is provided for the semantics and an eventual discourse mechanism.

(NON-QUANT NQ-CLASS)
appears on "non quantificational" **DETHEAD**s that can take **PREDET**s. **NQ-CLASS** is used to control the **PREDET** elements that appear with a given **DETHEAD**; it has the arguments:

(NUM-DET NUM-TYPE)
appears on "numerical" determiners; it has two values: **(CARD-NUM)** for cardinal numbers such as "three" and "twenty one" and **(FRACT-NUM)** for fractions, such as "three quarters", as well as the variable **:NUM-TYPE**. Various **PREDET**s can appear with any sort of numerical determiner, such as "more than" ("more than six ships", "more than three quarters of the ships"), while others can only appear with

the cardinal numbers, such as "all" ("all six ships", "???all three quarters of the ships").

(NON-NUM-DET NND-TYPE)
appears on "non numerical" determiners; it has two values: **(ART-DET)** for the traditional articles "the", "a", and "an" as well as the demonstratives "this", "that", "these", and "those" and **(POSS-DET)** for the possessive determiners, which include both "possessive pronouns" "my", "your", "their", etc. and full scale possessive NPs, as well as the variable **:NND**. Non numerical **DETHEAD**s are grouped together against numerical **DETHEAD**s because of co-occurence restrictions with the rest of NP, rather than because of co-occurence restrictions with **PREDET**; e.g. in constructions like "the/my last three vacations" (vs. "*six/three quarters last three vacations") and "the/his second trip" (vs. "*six/three quarters second trip").

:NQ the variable value for this feature.

:DET-CLASS
the variable value for this feature.

NBARFLAG specifies whether the determiner can be followed by a "null N-BAR" i.e. whether the head of the NP in which this determiner appears can be unrealized (as in "All the ships in Diego Garcia are C4 but those in Honolulu are C1'). This feature takes the argument **ADJUNCTP** which controls whether the determiner can appear followed by an NP adjunct in a null N-BAR construction. It has the values:

(ALLOWS-ADJUNCT)
for determiners that permit adjuncts in the null N-BAR construction, such as "those" ("those in the Indian Ocean").

(FORBIDS-ADJUNCT)
for determiners that take the null N-BAR construction but which do not allow adjuncts, such as "this" ("*this in the Indian Ocean").

:ADJUNCTP the variable value for this feature.

WH specifies whether the determiner is a "question" (or WH) determiner or not: elements like "which" and "how many" are **WH+** while ordinary determiners like "the", "a", "each", etc. are **WH-**.

PARTFLAG specifies whether the determiner is a partitive element, i.e. whether it can be followed by a partitive phrase introduced by "of" or not; compare "some of the ships", "each of the ships" with "*no of the men" (vs. "no men"), "*every of the men" (vs. "every man"). This feature has the value **(+PARTITIVE)** for determiners that can only appear followed by partitive phrases (such as "none"), **(-PARTITIVE)** for determiners that can never appear followed by partitive phrases (such as "no"), and **:PARTITIVE** for determiners that can optionally take a partitive phrase (such as "some").

2.5.6.2 Partitive Phrases and the Category PNP

Partitive phrases are introduced by the category **PNP** which has as its immediate dependents the **CASE-P** element "of" (see Section 2.5.9.1) followed by an NP. **PNP** takes the single argument **NUMBER** which agrees with the **NUMBER** argument of the NP it introduces.

2.5.6.3 Complements to Nouns

N-BAR is the constituent that contains the head noun of a noun phrase and its complements. There are currently only 12 subcategorization frames for nouns in the grammar; this reflects the nature of linguistic knowledge about complements to noun. While complements to nouns are less numerous than those to verbs, correspondingly less is known about them.

In addition to ordinary complements to nouns, N-BAR also contains a limited form of noun-noun compounding. This is also treated as a form of subcategorization; nouns that are allowed to be the head noun in a noun-noun modification structure bear the feature **TAKESPREMODIFIER**. This feature allows an N to take an N preceding it, as in "readiness rating" and "combat readiness"; it also allows this structure to recurse when the noun preceding the head noun itself has the feature **TAKESPREMODIFIER** as in "combat readiness rating". This feature allows for a subset of noun-noun modification, which is not currently handled in generality by the grammar, because of fears that allowing syntactically unrestricted noun-noun modification would result in explosive ambiguity.

2.5.6.4 Noun Phrase Conjunction

Currently, there are grammar rules for conjunction of two or more noun phrases with "and" or "or" as well as rules for conjunctions of the form "both NP and NP" and "either NP or NP". Currently, the pairs of conjoining elements (e.g. "both" and "and", "either" and "or", etc.) are introduced directly in the conjunction rules; in the future, it is likely that they will be dominated by a separate category, such as **CONJUNCT** and that there will be a separate category, such as **CO-CONJUNCT** for elements like "both", "either", "neither" that introduce conjunctions. Feature agreement between the two categories in the rule would ensure that the correct introductory element appeared with the correct conjunction, allowing there to be one general rule for such constructions, rather than one rule for each, as at present.

Schematically, this would work as follows:

```
((NP :NSUBCATFRAME (AGR :P (PLURAL)) (REALNP (NONUNITNP (-PRO (MISCNP))))
    :POSS :WH :CASEX
     :DET-STATE :DEF :DET-CLASS :SONS :TRX :TRX :CONTRACT (+CONJ))
 (CO-CONJUNCT :CONJ-TYPE)
 (NP :NSUBCATFRAME (AGR :PERSONX :NUMX) (REALNP :REALX) :POSS :WH :CASEX
    :DET-STATE
     :DEFX :DET-CLASSX :SONSX :TRX :TRX :CONTRACT (-CONJ))
 (CONJUNCT :CONJ-TYPE)
 (NP :NSUBCATFRAMEY (AGR :PERSONY :NUMY) (REALNP :REALY) :POSS :WH :CASEX
    :DET-STATE
     :DEFY :DET-CLASSY :SONSY :TRX :TRX :CONTRACT (-CONJ))
 (P-MIN :PERSONX :PERSONY :P))
((CO-CONJUNCT (CONJOINER))
 (both))
((CONJUNCT (CONJOINER))
 (and))
((CO-CONJUNCT (DISJOINER))
 (either))
((CONJUNCT (DISJOINER))
```

```
 (or))
((CO-CONJUNCT (NEG-DISJOINER))
 (nEither))
((CONJUNCT (NEG-DISJOINER))
 (nor))
```

Conjunction of NPs with "and" creates some interesting problems for the **AGR** feature. The number of a conjoined NP of this type is always **(PLURAL)**, regardless of the number feature(s) of the individual conjoined NPs. The person of a conjoined NP, however, *is* based on the person of the conjoined NPs. If one of the NPs is **(1ST)** person, the entire NP is **(1ST)** person: "John and I like ourselves/*themselves": if one of the NPs is **(2ND)** person and the others are all **(3RD)**, the entire NP is **(2ND)** person: "You and Bill like yourselves/*themselves". To capture this fact, the conjunction rule for NPs contains a dummy category, **P-MIN** that, in effect, performs a computation to determine the correct person of the conjunct. **P-MIN** takes three arguments, each of which is **PERSON**. The first of them is the **PERSON** feature of the first conjunct, the second is the **PERSON** feature of the second conjunct[25], and the third is the **PERSON** feature that is passed up as the **PERSON** feature of the entire conjunction. Here is the entire set of **P-MIN** rules that ensure the correct person on conjoined NPs.

```
;; 1st person always takes precedence
(((P-MIN (1ST) :P (1ST)))
 semantics (lambda () dontcare))
;; 2nd person yields to 1st
(((P-MIN (2ND) (1ST) (1ST)))
 semantics (lambda () dontcare))
;; 2nd person takes precedence over 3rd
((P-MIN (2ND) (3RD) (2ND)))
 semantics (lambda () dontcare))
; 2nd person and 2nd person => 2nd person
(((P-MIN (2ND) (2ND) (2ND)))
 semantics (lambda () dontcare))
;; 3rd person always yields
((P-MIN (3RD) :P :P))
 semantics (lambda () dontcare))
```

2.5.6.5 Specialized Noun Phrase Constructions

In addition to the general noun phrase constructions already discussed, there are also specialized noun phrase constructions. Currently, these include:

time expressions in any of the following forms:

fifteen hundred
fifteen hundred hours
fifteen hundred hours zulu
fifteen hundred zulu

Time expressions may also be introduced by prepositions, such as "at".

[25]Conjoined NPs with more than two conjuncts are handled as right branching structures with only two conjuncts on each level.

date expressions in any of the following forms:

twenty December
December twenty
December twentieth
twentieth December
twentieth of December
the twentieth December
the twentieth of December

as well as date expressions consisting only of the day of the week, e.g. "Wednesday". Date expressions may also be introduced by prepositions, such as "on".

year expressions in either of the forms:

nineteen eighty seven
eighty seven

latitude and longitude expressions
in any of the following forms:

one hundred degrees north fifty degrees east
one hundred degrees north and fifty degrees east
one hundred degrees fifty minutes north and fifty degrees forty five minutes east
one hundred degrees fifty minutes north fifty degrees forty five minutes east
a latitude of one hundred degrees north and a longitude of fifty degrees east
latitude one hundred degrees north longitude fifty degrees east
latitude one hundred degrees north and longitude fifty degrees east

2.5.7 Adjective Phrase Constructions

Adjectives may appear alone and may also be modified by a special class of adjectival specifiers: such as "very" or "how". Other, more specialized Adjective Phrase constructions are also handled:

<ordinal number> <superlative adjective>
e.g. "second best"

<unit NP> <comparative adjective>
e.g. "three miles closer"

<comparative adjective> than NP
e.g. "faster than the Wasp"

<comparative adjective> than S
e.g. "faster than the Wasp is"

<comparative adjective> PP *than* NP *is* PP
e.g. "farther from Guam than the Wasp is from Diego"

<comparative adjective> than NP PP
e.g. "closer than the Wasp to port"

less <positive adjective> than NP
e.g. "less truthful than John"

less <positive adjective> than S
e.g. "less truthful than John is"

as <positive adjective> as NP
e.g. "as fast as the Wasp"

as <positive adjective> as S
e.g. "as fast as the Wasp is"

2.5.7.1 Complements to Adjectives

Like verbs and nouns, adjectives also take complements. The complexity of the complement system to adjectives is between that of nouns and verbs. Currently, there are only 8 subcategorization frames, but others will be added. As was the case with the complement system to nouns, there are fewer complement types to adjectives, but less work has been done in this area. As was true of NP and N-BAR, ADJ-BAR is the category that dominates the head ADJ of an ADJP and its complements. The complete set of subcategorization features for ADJs will not be discussed here, but two will be mentioned.

(TAKESPRENOUN) appears on ADJs that take a bare N before them, such as "harpoon capable". It is an open question as to whether such constructions should be handled this way or not. However, this construction does seem to behave like an independent subcategorization frame, rather than as a morphological compounding process, in that it does seem to preclude other complements. Compare "capable of destroying the city" vs. "*harpoon capable of destroying the city".

(TAKES-UNIT) appears on ADJs that allow a unit NP to appear before them, such as "three miles long". The unit NP is actually introduced under ADJP, rather than under ADJ-BAR, since the unit NP seems to be incompatible with other ADJP level specifiers: "*very three miles long", "*more three miles long", "*as three miles long". This is in contrast to the **(TAKESPRENOUN)**, which displays no such incompatibility: "quite harpoon capable", "more harpoon capable", "as harpoon capable".

2.5.7.2 Adjective Phrase Position

Adjective phrases can appear in three different positions (excluding constructions where they appear as complements): pre-nominally, as in "the C3 ship"; post-nominally, as in "the fuel remaining"; and in predicate position, as in "the ship is C3". Most adjectives can appear in pre-nominal and predicate position (as "C3" illustrates) but there are some adjectives that can only appear predicatively ("the man is afraid" vs. "*the afraid man") or post-nominally ("presents galore" vs. "*galore presents" or "*the presents are galore"). In addition, even adjectives that can normally appear pre-nonimally appear post-nominally when they occur with a complement; compare "a faithful servant", "a servant faithful to his master", "*a faithful to his master servant".

The feature **POSITFLG** is used to control the placement of ADJPs. This feature has the following values:

(PRENOMADJ) appears on ADJPs headed by ADJs that can appear pre-nominally. ADJs that can only appear in this position have this feature specified in the lexicon; ordinary ADJs have this feature inserted in the virtual rules created by the morphology.

(NOT-PRENOMADJ ADJPOS)
appears on ADJPs headed by ADJs that can appear other than in pre-nominal position. **ADJPOS** specifies whether this is post-nominal or predicative. Its values are:

(PREDADJ) appears on ADJPs headed by ADJs that can appear predicatively. ADJs that can only appear in this position have this feature specified in the lexicon;

ordinary ADJs have this feature inserted in the virtual rules created by the morphology.

(POSTNOMADJ) appears on ADJPs headed by ADJs that can appear post-nominally. Normally, this feature is lexically specified. The grammar also inserts this feature on ADJPs that contain ADJs with overt complements.

:ADJPOS, **:ADJPOSX**the variable values for this feature.

:POSIT, **:POSITX**
the variable values for this feature.

2.5.7.3 Adjective Phrase Conjunction

Currently, there are grammar rules for conjunction of two or more adjective phrases with "and" or "or".

2.5.8 Adverbial Constructions

Adverb Phrase (ADVP) is currently the most impoverished category in terms of internal structure. ADVP has the following arguments:

(ADVP ADVTYPE ADVPOS WH TRACE TRACE)

WH and **TRACE** are familiar; **ADVTYPE** specifies the type of the ADVP and **ADVPOS** specifies the position in which it may appear. The values of **ADVTYPE** are:

(TIMEADV) introduces temporal ADVPs such as "now" and "when" as well as temporal NPs.

(DATEADV) introduces date NPs.

(MANNERADV) currently only introduces "how".

(REASONADV) currently only introduces "why".

:ADVTYPE the variable value for this feature.

ADVPs can appear in three positions in an S: initially, as in "Currently, the Eisenhower is in Diego."; medially, as in "Eisenhower never went to C5."; and finally, as in "Eisenhower is in the Indian Ocean now." Since not all ADVPs can appear in all three of these positions, the feature **ADVPOS** is used to specify where an ADVP can appear. The values of **ADVPOS** are:

(S-PERIPHERAL S-POSITION)
appears on ADVPs that can only appear at the "periphery" of an S: i.e. initially or finally. **S-POSITION** controls which of these two positions an ADVP appears in.

(S-INITIAL) specifies that the ADVP appears only S initially.

(S-FINAL) specifies that the ADVP appears only S finally; e.g. "yet". The ADVP rules for ADVP traces also specify **(S-FINAL)** as the source of the trace, to prevent ambiguity.

:S-PERIPHERAL the variable value for this feature; specifies that the ADVP appears either S initially or S finally; e.g. "today".

(VP-INITIAL) appears on ADVPs that can only appear medially in an S, before the VP, such as "ever".

:ADVPOS the variable value for this feature; specifies that the ADVP appears S initially, S medially, or S finally; e.g. "now".

Currently, ADVPs are only introduced in S medial and S final position; the S initial position for ADVPs will be added later.

In terms of internal structure, ADVPs can superficially contain an Adverb, without any modifiers, and may also be modified by a special class of adverbial specifiers such as "very" or "how". Other, more specialized Adverb Phrase constructions are also handled:

<unit NP> <comparative adverb>
e.g. "four days sooner"

the *<superlative adverb>* of NP
e.g. "the longest of submarines which are in Bering Strait"

<superlative adverb> of NP
e.g. "longest of submarines which are in Bering Strait"

In addition to introducing adverbs, ADVPs also introduce the NPs for time, date, and year expressions discussed in Section 2.5.6.5. LOC-TEMP-PHRASEs appear as dependents of ADVP, rather than as direct dependents of OPTSADJUNCT in sentence final position, for semantic purposes.

2.5.8.1 ADV-BAR Rules

Currently, there are no ADV-BAR rules. Adverbs are introduced directly in the grammar as daughters of ADVP at present. However, ADV-BAR is a declared category of the grammar to allow ADVs to eventually take complements, like V, N, and ADJ, since there are complement taking adverbs, such as "especially" and "such as".

2.5.9 PP and Related Categories

While the current grammar contains the category PP (Prepositional Phrase), its presence is temporary. Various linguistic researchers, have pointed out problems with the category PP[26] and there seem to be good reasons, both syntactic and semantic, for decomposing PP into separate categories.

- The external distribution of the elements traditionally labelled as prepositions is radically non-uniform. There are various positions in which prepositional phrases are said to appear, including NP adjunct position, S adjunct position, and predicate positions, but not all prepositions can appear in all these positions.
- The internal structure of prepositional phrases also differs wildly. Some, but not all, can allow pre-modification by unit NPs, may appear "intransitively" (i.e. without NP complement), may recursively take other PPs as complements, etc.
- With regard to pronominal reference and related phenomena, it has been noted that while some prepositions behave as if they head a branching category, so that their NP complements do not c-command outside the PP, others behave as if they did not introduce a category dominating their purported NP complements, since the NP can c-command outside the PP.
- Parallel to this are semantic interpretation facts. Some prepositions do play a role in semantic interpretation: their semantic translation is applied to that of their NP complement and, in turn, the

[26]For example, Hoffman [22] argues for the existence of a separate category of Locative Phrase.

resulting translation is passed on. Other prepositions act as if they were "invisible": they merely pass on the semantic translation of their NP complement, without contributing anything to the meaning of the PP, as a whole.

For all these reasons, we have undertaken to decompose PP into distinct categories. However, this decomposition is not yet complete, and the category PP exists in the current grammar alongside some of the categories that will eventually replace it.

PP and its sister categories all have the same basic structure:
<preposition> **NP**

In addition to PP, there are currently also CASE-PHRASEs and LOC-TEMP-PHRASEs.

2.5.9.1 CASE-PHRASE Rules

Linguists, going back at least as far as Zellig Harris, have noticed that some "prepositions" behave analogously to Case markers in inflected languages such as Latin, Greek, Finnish, or Hungarian. Syntactically, the "prepositions" that appear in this category seem to be characterized by an inability to take various modifiers that other "prepositions" do; semantically, "prepositions" that head CASE-PHRASES do not seem to contribute to the semantics of these phrases, which, instead, have the semantics of their NP "complements". This is natural, of course, if these "prepositions" are analogous to Case markers, rather than to lexical heads.

2.5.9.2 LOC-TEMP-PHRASE Rules

LOC-TEMP-PHRASE roughly corresponds to Hoffman's [22] Locative Phrase category. "Prepositions" in this category are characterized by the modifiers that they take (e.g. "right" and unit NPs), the positions that they appear in (NP and S adjunct positions), and by the fact that they semantically contribute to the meaning of the phrases that they head. This category is called LOC-TEMP-PHRASE since the elements that head it are interpreted either as locative or temporal elements: some are interpreted ambiguously as either.

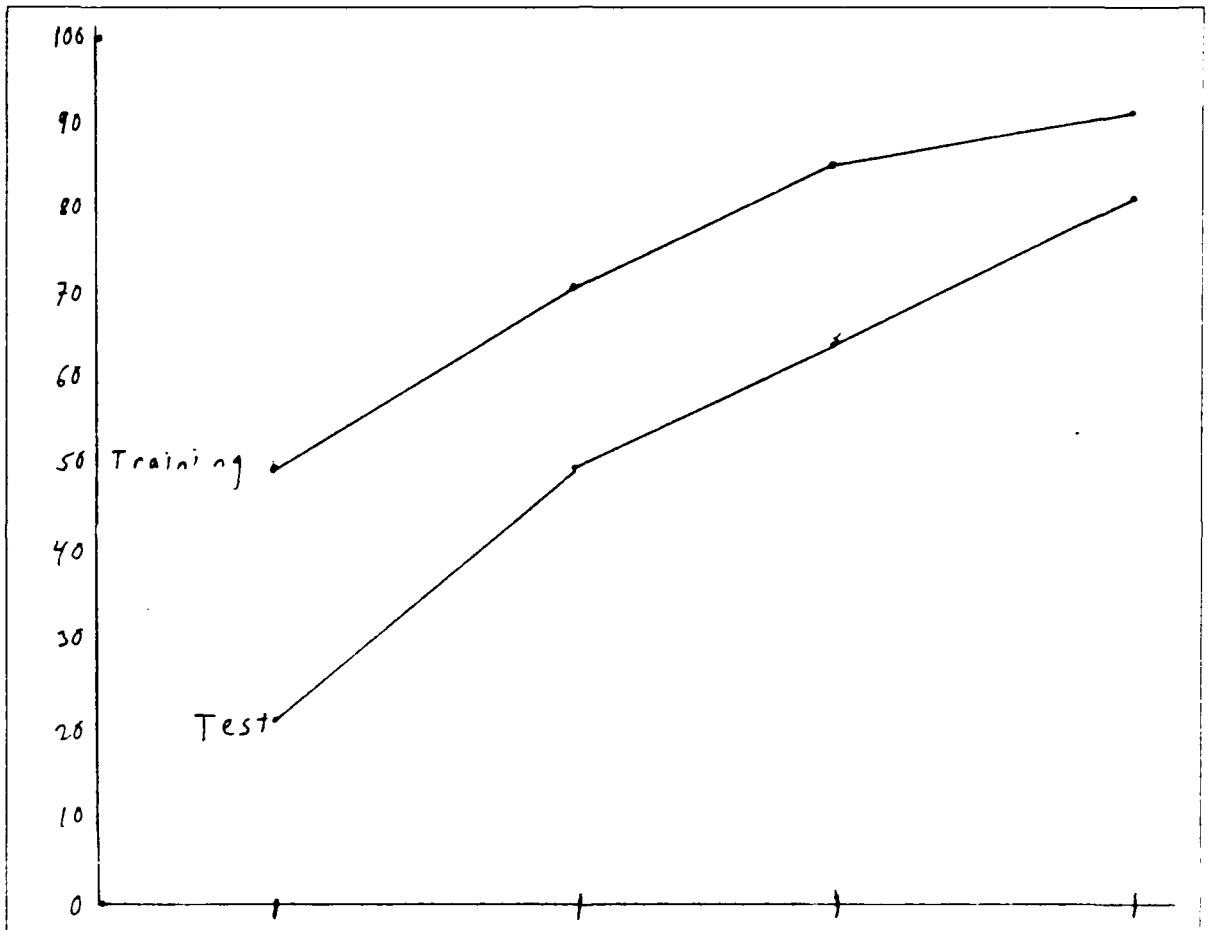
2.6 Quantitative Measures of Coverage

2.6.1 Grammar Size

The current ACFG grammar contains 866 rules: of these, 424 introduce grammatical formatives (such as the articles "a", "the", prepositions, etc.) The remaining rules handle the grammatical constructions of the language.

2.6.2 Syntactic Coverage

Syntactic coverage is measured using sentences from the Resource Management database corpus. As was noted in Section 1.4 ,these sentences are divided into two subsets: a training corpus of 791 sentences and a test corpus of 200 sentences. The grar .mar currently covers 91 percent of the training corpus and 81 percent of the test corpus. Figure 2-5 shows the figures for training and test corpus coverage for July, 1987, October, 1987, and February, 1988, as well as the current coverage.



	July, 1987	October, 1987	February, 1988	December, 1988
Training	50%	71%	85%	91%
Test	22%	50%	64%	81%

Figure 2-5: BBN ACFG Resource Management Training and Test Corpus Coverage

2.6.3 Perplexity

For spoken language systems, *perplexity* is an important measurement. Perplexity is defined as follows:

$$Q = 2^H$$

where Q is perplexity and where H is

$$H = \frac{-1}{n} \log P(w_1, \ldots, w_n)$$

where $w_1, \ldots, w_n$ represents a set of possible input sentences stacked as an n-long text (typically from a corpus). Intuitively, given some prefix (initial substring, possibly null) of a sentence, perplexity represents the number of possible word choices that are allowed to follow within a given grammar.

The perplexity of the current grammar is approximately 700, which is very near the total size of the terminal vocabulary. Nevertheless, as Section 4.5 shows, performance of the spoken language system as a whole is not impaired by this high perplexity. There may be several reasons for this somewhat puzzling result. First, as is suggested in Section 4.5, it may be that the grammar constrains the appearance of function words, which helps the performance of the overall system, since function words are typically hard to recognize and can degrade overall recognition performance. Another possibility is that perplexity is not as useful a measure in systems involving a generative grammar as in those involving bigram or trigram language models. While it is possible, in the latter types of language models, to predict all the permissible following words at any point in an input string, it is not clear that it is possible to do so in a grammar that includes recursion and iteration.

Consider the NP grammar sketched out in Section 2.5.6. If the parser finds a plural determiner, signalling the start of an NP, the next word may be any one of the set of comparative, superlative, or positive adjectives, any progressive or passive participle (since these may be treated as adjectives), or any noun. In fact, if the next word is a noun, it cannot even be ruled out if it is not plural, since it might be part of a noun noun compound; while the head of this compound would need to agree with the determiner in number, any preceding noun need not; until the parser is sure the head noun has been found, no agreement constraints can be checked. However, when the end of the NP is found, either by encountering a constituent that cannot be part of the NP, such as a finite verb, or simply running out of input, certain constraints that have been deferred, as it were, such as agreement between the determiner and the head noun, can be imposed. Thus, perplexity, which measures the predictive power of a language model, rather than the a posteriori constraints that it imposes, may paint a picture of a generative grammar as being looser than it really is.

2.6.4 Ambiguity

As the coverage of the grammar increases, two types of ambiguity also increase. First, at any given point in a sentence, there will typically be more possible choices that must be tried. Second, the number of parses that a sentence has may increase. Information on the number of parses per sentence has been collected for the 726 training and 162 test sentences that the current grammar handles.

These figures show the following distribution:

Corpus:	Mean	Median	Mode
Training	4.26	2.65	1
Test	4.70	2.72	1

There are various reasons why a sentence may be treated as ambiguous by the parser. First, it may be the case that a reasonable grammar for English will produce multiple syntactic interpretations for a given sentence, though semantic or pragmatic information may decide among them. This type of ambiguity might be considered irreducible. Another type of ambiguity may simply be the result of errors in the grammar, either because a syntactic phenomenon has not been handled correctly or because of a typographic error. Such mistakes are corrected when detected. The third source of ambiguity is more problematic. These are cases where the formalism in which the grammar is written introduces certain types of ambiguity that cannot be eliminated by rewriting existing rules or adding new rules. For examples, noun compounds such as "data screen" are *n* ways ambiguous, where *n* is the number of possible grammatical number readings of the first member of the nominal compound (for example, "data" is treated as both singular and plural in the current grammar). In some cases, it may be possible to eliminate the ambiguity by changes to the formalism. For example, in the present case, the addition of a disjunction mechanism, which would collapse all the separate readings into one, would probably do the job.

Our experience has been that while the average number of parses per sentence is usually quite reasonable, in cases of conjunction or ellipsis the number of parses can grow wildly. In order to obtain broad coverage without explosive ambiguity, we have experimented with a version of the parser in which rules are sorted into different levels of grammaticality. In this version of the parser, parses are ranked according to the rules utilized. Initial efforts, in which ranks were assigned to rules by hand, are encouraging. A slighlty expanded version of the grammar (from 866 to 873 rules) including rules that increased ambiguity, such as determiner ellipsis, had an average of 17 parses per sentence and a mode of 2 parses for the training corpus. However, when only first order parses were considered the average was 3 parses and the mode was 1. Similar results were obtained for the test set. These results are shown in tabular form below.

Corpus:	Mean	Median	Mode
Training (All Parses)	17.33	4.31	2
Training (Highest Ranked Parses)	2.92	2.50	1
Test (All Parses)	7.40	3.00	2
Test (Highest Ranked Parses)	3.41	1.50	1

Coverage with this version of the grammar was 94% on training and 88% on test.

We have also experimented with utilizing statistical methods of assigning rules to levels, based on the frequency of occurence of the rules of the grammar in the training corpus. Testing the results of this probabilistic grammar against a corpus of 48 sentences (from the training corpus) that were manually parsed, the parse assigned the top score by the probabilistic parser was correct 77% of the time. Looking only at the top 6 parses, the correct parse was present 96% of the time. Since the success rate is 96% considering only the top 6 parses, while 50% of the sentences have 6 or more parses with this version of the grammar, this suggests that this probabalistic approach is on the right track.

We hope that use of a parser that finds all parses for a given input utterance, in conjunction with a grammar whose rules are annotated with either hand-assigned scores or automatically assigned probabilities, will enable us to treat truly ambiguous utterances as ambiguous, while not creating spurious ambiguity.

2.7 Future Plans

This section describes the work that is currently planned to increase the coverage and performance of the syntactic portion of the system.

2.7.1 Integrating the Parser with Ranked Rules

Since the program for collecting coverage information for the training corpus also provides information about the number of parses per sentence, it is possible to note sentences with a large number of parses, examine the parse trees assigned to them, and modify the grammar to reduce the number of parses, where the grammar is overgenerating. At times, this has resulted in dramatic savings: "Why was Citrus's MOB mission area changed thirty one September" had its total number of parses reduced from 54 to 2 with changes in two grammar rules. While this process will undoubtedly continue in the future, it is labor intensive, since all the parses for an ambiguous sentence must be examined by hand, so we have sought more general procedures that can reduce ambiguity but which are also linguistically motivated. The most promising scheme, which we have already begun exploring, as reported in the previous section, is to assign probabilities to grammar rules. Once this work is more mature, we can explore the ways in which a parser making use of this information can be integrated into the general architecture of the system:

1. Find all parses for a given input, but ignore all parses that use grammar rules with a probability below a fixed threshold.
2. Find all the parses of a given input that involve only grammar rules with a probability above a fixed threshold; parses involving rules with probabilities below this threshold will not be constructed.
3. Use probabilities to arrange the rules of the grammar into tiers; all parses for a given input involving rules from all tiers will be found, but parses involving rules from lower probability tiers will be ignored.
4. Use probabilities to arrange the rules of the grammar into tiers; in parsing a given input, if at least one

parse is found using rules from the tier with the highest probability, parses involving rules from tiers with lower probabilities will not be constructed. However, if no parse is found at the highest tier, rules from successively lower tiers will be used, until at least one parse is found.

2.7.2 Changes to the Grammar Formalism

As one more technique for eliminating spurious ambiguity, we will explore the possibility of using a simple disjunction mechanism to collapse the readings assigned to noun-noun compounds, discussed above in Section 2.6.4.

3. The Semantic Component

3.1 Introduction

This chapter presents the semantic component of the Spoken Language System. The semantic component operates upon the output of the syntactic component in such a way as to give the correct response to a user's request. This introductory section presents the overall architecture and design philosophy of the semantic component.

Subsequent sections of the chapter are as follows:

- Section 3.2 discusses the structural-semantic rules of the system.
- Section 3.3 presents the lexical semantic component.
- Section 3.4 discusses the methods of logical expression simplification used in the system.
- Section 3.5 presents the answer-retrieval component of the system.

3.1.1 System Architecture

As can be seen from the Figure 1-1 semantic processing of an input query takes place in stages. First, the output of the parser, the parse tree, is passed to the structural semantic module. This module deals only with the semantic consequences of syntactic structure. It uses a set of structural semantic rules, which are paired one for one with the syntactic rules of the grammar, and a collection of lexical semantic entries, paired one-for-one with the words in the lexicon. Using a simple compositional algorithm that works recursively on the parse tree, the module constructs an expression of the logical language EFL (for English-oriented Formal Language). EFL is a somewhat unusual logical language in that its expressions can be ambiguous, reflecting the (possible) ambiguity of English words.

The next stage of semantic processing concerns itself with lexical semantics. It accepts as input an expression of EFL and returns as output zero or more expressions of the logical language WML (for World Model Language). WML is an unambiguous logical language which includes one descriptive constant for each primitive category or relation of the subject domain. This EFL to WML translation component uses a set of local EFL to WML translation rules, which associate one or more expressions of WML with each constant of EFL. The algorithm for translation includes a filtering component, which removes from the set of final translations semantically incompatible combinations of WML expressions.

The third stage of semantic processing takes an expression of WML and outputs an expression of another logical language, DBL (for DataBase Language). DBL includes a constant symbol for each data file of the database. Finally, the fourth stage of processing, evaluation, computes the "value" of the DBL expression against the actual contents of the data files in the database. This "value", which constitutes the answer to the original input question, is expressed in yet another (much simpler) language, CVL (for Canonical Value Language). CVL has a very small set of operators and only rigidly denoting symbols as constants.

3.1.2 The Logical Languages

Each of the logical languages just mentioned—EFL, WML, DBL and CVL—is derived from a common logical language [47] from which each differs only in the particular constant symbols which are allowed and not in the operators (the only exception is CVL, whose operators are a *subset* of the operators of this common language).

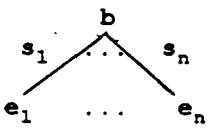
This logical language has three main properties. First, it is *higher-order*, which means that it can quantify over sets, tuples and functions instead of just over individuals. Second, it is *intensional*, which means that the denotations of its expressions are assigned relative to external *indices* of world and time. It incorporates an "intension" operator which denotes the "sense" (with respect to these indices) of an expression. Third and finally, the languge has a rich system of *types* which serve to delimit the set of meaningful expressions in the language.

Expressions of the logic are divided into three groups: constants, variables and complex expressions. Complex expressions are built up by the formation rules from constants, variables and other complex expressions.

Expressions of the logic are labeled trees in which both nodes and branches are labeled. Constants and variables are the terminal nodes. The node labels are called "branching categories", the branch labels, "selectors". Each branching category 'b' is associated with a set of selectors 'F-SELECTORS(b)'. If 'b' is a branching category, $e_1,...,e_n$ expressions, and $\{S_1,...,S_n\}$ = F-SELECTORS(b) then the following is an expression:

$$\langle b,\ \{\langle S_1, e_1\rangle, \ldots, \langle S_n, e_n\rangle\}\rangle$$

or drawing it in tree form:



An example would be the branching category APPLY whose selectors are the set {FUNCTION,ARGUMENTS}. Let 'FREDERICK' and 'READINESS-OF' be constant symbols. Then the following is an expression:

```
<APPLY, {<FUNCTION, READINESS-OF>, <ARGUMENT, FREDERICK>}>
```

Certain branching categories have a special selector, 'VAR', as one of their selectors. Let 'FORALL' be a branching category whose selectors are {VAR,RESTRICTION,FORMULA} and let 'X' be a variable, 'P' and 'SHIPS' constants. Then the following represents an expression which is said to bind the variable 'X':

```
<FORALL, {<VAR, X>, <RESTRICTION, SHIPS>,
          <FORMULA, <APPLY, {<FUNCTION, P>,
                             <ARGUMENT, X>}>>}>
```

Obviously this structural notation is cumbersome to write. Therefore, an external form of the language is provided. The above would be written:

```
(FORALL X SHIPS (P X))
```

Each expression of the logic has a *type*, which constrains both the types of other expressions with which the expression can meaningfully combine and the values the expression can assume. There is an infinite set of types,

enumerated by a set of recursive rules. The base of the recursion is a finite set of atomic types, varying with application domain but including at least the set INTEGERS, REALS, WORLDS, TIMES, STRINGS, SPEECH-ACTS, TRUTH-VALUES, and NULL-SET. For reasons that we shall see shortly, these are called the *formal* types.

The following are some of the formation rules for types. We assume the following:

every atomic type is a type

if α, α_1,...,α_n and β are types then the following are also types:

SETS(α)
FUN(α,β)
TUPLES(α_1,...,α_n)
UNION(α_1,...,α_n)

Types are assigned "domains", which are sets of semantic values representing the possible denotations which expressions having that type may take on. Formal types are special in that their domains are fixed for all allowed interpretations of the language. The domains of atomic types are mutually disjoint. The relation SUB-TYPE? and the binary operation TYPE-INTERSECTION are defined for types. Note that for distinct atomic types α and β, TYPE-INTERSECTION(α,β) = NULL-SET.

As an example, '1' and 2' are constants of type INTEGERS, while '2.0' is a constant of type REALS and 'TRUE' and 'FALSE' are constants of type TRUTH-VALUES. The symbol '+' is a constant of type

```
FUN (TUPLES (UNION (REALS, INTEGERS) , UNION (REALS, INTEGERS) )
     UNION (REALS, INTEGERS) )
```

i.e., a function type.

The types of complex expressions are computed by a rule associated with their branching category, a rule which takes as input the types of the sub-expressions at their branches. Let the symbol 'FREDERICK' be a constant of type 'SHIPS' and let 'READINESS-OF' be a constant of type FUN(SHIPS,R-VALUES), where 'R-VALUES' is also an atomic type. Then the construction:

```
<APPLY, {<FUNCTION, READINESS-OF>, <ARGUMENT, FREDERICK>}>
```

is a meaningful expression of type 'R-VALUES'. Now, suppose 'RONALD-REAGAN' is a constant of type 'PERSONS'. The construction

```
<APPLY, {<FUNCTION, READINESS-OF>, <ARGUMENT, RONALD-REAGAN>}>
```

is not a meaningful expression, and has the type NULL-SET. It can never have a denotation under any indices, and accordingly has the empty set as the set of values it can take on.

We have now put into place the machinery needed to distinguish between meaningful and meaningless expressions. Meaningless expressions have NULL-SET as their type, and can never have a denotation at any index. Meaningful expressions may fail to have a denotation at certain indices—consider "The King of France"—but nonetheless have a denotation at *some* index.

The infinite hierarchy of types allows more complex functions than we have seen so far—functions on sets

for instance. This power turns out to be necessary for many English utterances. Consider a sentence like "The boys gather". Unlike the similar sentence "The boys walk", it is not true that each boy, individually, gathers—this simply wouldn't make sense. Rather, it is a predication that one makes of the set of boys as a whole. To represent it, one needs a predicate which is applied to sets of people, i.e. a function constant of type:

```
FUN (SETS (PEOPLE) , TRUTH-VALUES)
```

If 'GATHER' is a function constant of this type, then the utterance "The boys gather" can be represented as:

```
GATHER (BOYS)
```

This is a small illustration of the power and flexibility of the infinite type system of the language. Examples still more complex than the verb "gather" do not give us trouble. If we like, we can have constants whose types are sets of sets of sets, or functions from functions to functions.

So far we have been content to simply introduce a new constant whenever we have need. This procedure quickly becomes inconvienent when we want to construct the representation of a notion which is built up from others. Consider a function constant 'LOVES' whose type is:

```
(FUN (TUPLES PEOPLE PEOPLE) TRUTH-VALUES)
```

and an individual constant 'MARY' whose type is 'PEOPLE' and which translates the name "Mary" in our lexicon. Now suppose we want to consider a special property, the property of loving Mary. We wouldn't want to have to come up with a new constant symbol, say 'LOVES-MARY', for every possible object of affection in the domain of discourse. What we really would like to do is have some way of constructing an expression representing this property out of materials—constant symbols—already to hand.

The branching category 'LAMBDA' gives us a way of doing this. Its selectors are {VAR,RESTRICTION,BODY}. The property of loving Mary is then expressed by:

```
(LAMBDA X PEOPLE (LOVES X MARY))
```

The type of this expression is a function type from the type of its VAR, or bound variable, (PEOPLE) to the type of its BODY (TRUTH VALUES). This is simply a predicate.

Now suppose we wanted to say that a particular person, say Bill, loves Mary. Let "Bill" be translated by an individual constant 'BILL' of type PEOPLE. We can then apply the above LAMBDA expression as we would any other predicate, to obtain the expression:

```
((LAMBDA X PEOPLE (LOVES X MARY)) BILL)
```

This expression can be simplified to an equivalent expression by a process called Lambda-reduction. If the argument (here, 'BILL') is of a type "appropriate" to that of the bound variable of the lambda-expression, it can be substituted for that bound variable in the body of the lambda-expression. (The rule is actually more complicated, renaming free variables in the argument that would otherwise become bound in the substitution, and not substituting for variant occurences of the bound variable that are bound again in the body.) The above can be reduced to:

```
(LOVES BILL MARY)
```

The LAMBDA branching category plays a very important role in our system. The translation rules that map

between the EFL,WML and DBL levels use lambda-expressions to construct representations in the target language that are equivalent to notions in the source language. In this way a mapping can be constructued between expressions of the two languages even though they have different sets of constant symbols.

3.1.3 The Semantic Framework System

3.1.3.1 Introduction

The semantic framework system implements the abstractions of our logic in software. It is an interconnected system of modules and definitions which allow other programs—specifically the various translation modules sketched in Section 3.1.1—to be written in terms of these mathematical abstractions without concern for implementation details. In consequence the translation modules themselves are quite simple, and comprise only a few pages of code. Translation modules and indeed the entire processing architecture can be quite easily modified as the need may arise.

The semantic framework system includes:

- Data structure support for logical language expressions
- A means for easily extending the logic to include new operations
- Declaration functions for specifying descriptive constants
- A type system to compute expression types
- A sub-system for performing translations and transformations on expressions
- Syntax and consistency checkers to prevent errors in data entry

We now proceed to discuss each of these in detail.

3.1.3.2 Logical Expressions as Data Abstractions

We have implemented the logical language with LISP functions in which branching categories and selectors are represented as LISP atoms and constructions as implementation-dependent data-structures.

We have the function CONSTRUCTION:

CONSTRUCTION (*branch-category branches*)

where the argument *branches* is an association list pairing selectors and expressions or, in the case that the selector is multi-branching, LISP lists of expressions. It creates and returns a tree data structure. The following constructs a universal quantification expression which claims for all CRUISERS that the predicate P holds:

```
(CONSTRUCTION 'UNIVERSAL-QUANTIFICATION
              '((FORMULA APPLY P (VARIABLE X SHIPS))
                (VAR VARIABLE X SHIPS)
                (RESTRICTION . CRUISERS)))
```

We have the function BRANCH-CATEGORY:

BRANCH-CATEGORY (*expression*)

which takes a tree and returns its branching category. If it were applied to the example just presented it would return the atom UNIVERSAL-QUANTIFICATION.

There is the function SELECTION:

SELECTION (*expression selector*)

which takes a tree data structure and appropriate selector, and returns the sub-tree structure pointed to. Suppose the construction returned above to be bound to the atom FOO. Then the following:

```
(SELECTION FOO 'VAR)
```

would evaluate to '(VAR X SHIPS)'.

Also available are functions (actually macros) which evaluate only some of their arguments. These are very handy when the selectors and branch category of the expression are already known when code is written. There is the function MAKE, whose form is:

```
(MAKE <bc>
      <sel1> <exp1>
          .....
      <seln> <expn>)
```

where only the <expi> are evaluated. There is also the function SELECT, whose form is:

```
(SELECT <selector> <exp>)
```

where only <exp> is evaluated.

3.1.3.3 Functions for Defining Constants

The following functions are used to define constants and types:

dt (*name language &optional def*)
dft (*name*)

dc (*name language type*)
dfc (*name type*)

The above stand for, respectively, "declare type", "declare formal type", "declare constant", "declare formal constant". The *name* and *language* arguments can only be filled by LISP symbols.

The *language* argument must be supplied for non-formal types and constants, and must be a declared language name. The list of language names is bound to the atom **language-names** To add a language name, one adds to this list.

3.1.3.4 Functions for Extending the Language

The semantic framework system has a unique and powerful feature: it directly supports the extension of the logical language used for semantic representation.

Branching category declarations are separated into two classes: those declaring regular branching categories and those declaring branching categories of the type sub-language. The following are their argument patterns:

DEF-BC (*branch-category selector-list typerule &optional eval-rule*)

DEF-TYPE-BC (*branch-category selector-list &optional eval-rule*)

None of these arguments are evaluated. The *branch-category* argument must be a LISP atom, and the *selector-list* argument a list of atoms. This argument specifies the value of the function F-SELECTORS for the given branching category.

The *typerule* argument specifies a rule for computing the type of the expression of the branching category from the types of its sub-trees. This rule is a LISP lambda-expression with the same number of arguments as there are selectors for the branching category. Type rules which impose requirements on the input types use the function COMPATIBLE?. This function takes two arguments: the type required and the actual type. Its definition varies according to the value of the global switch **TYPE-CHECK-MODE**. If the value is **LOOSE**, type-intersection is used; if the value is **STRICT**, sub-type? is used.

The *eval-rule* argument is also optional, and analogously specifies a rule for computing the *value* of the expression from the values of its sub-trees. Eval-rules are LISP code around which has been wrapped either a form '(EVAL-T *)' or a form '(EVAL-O *)'. "EVAL-T" stands for "evaluate transparently". A rule of this kind is supplied with the values of the sub-trees of the expression. "EVAL-O" stands for "evaluate opaquely". A rule of this kind is supplied with the sub-trees themselves, and must itself take on the responsibility for how they are evaluated. EVAL-O rules are inherently non-compositional, and are (currently) used only for branching categories which bind a variable.

3.1.3.5 Translations and Transformations

A number of functions are provided that facilitate the transformation of one logic expression into another.

We distinguish two different kinds of transformations: local and global. In a local transformation, only constant symbols are transformed. Branch-categories are left unchanged and thus the structure of the input expression carries over to the output expression. An example of this kind of transformation would be the WML to DBL translation.

The function TRANSLATE:

TRANSLATE (**exp ct**)

takes an expression and a function. The function argument takes a constant and returns an expression (whether

another constant or a complex lambda-expression) which is that constant's translation. The algorithm is quite simple and is given below:

```
TRANSLATE(exp CT) =def
   BC <- BC(exp)
   Selectors <- F-selectors(BC)
   (if: BC = CONSTANT
    then: CT(exp)
    elseif: BC = VARIABLE
    then: exp
    else: (construction BC
                        (pairlis selectors
                                 (for s in selectors
                                      collect
                                      TRANSLATE(selection(exp,s),CT)))))
```

Global transformations are unconstrained. They take the form '<input pattern> => <output pattern>'. Patterns are implemented by the notion of meta-expressions, distinguished from regular expressions by the appearance of meta-variables, which are atoms prefixed by "$". An example global transformation would be:

```
(ELEMENT-OF $x (SETOF $a)) => (EQUAL $x $a)
```

which states that that a term "$x" is an element of a one-element set whose element is a term "$a" if and only if "$x" is equal to "$a". The arrow indicates that the expression on the left-hand side is to be rewritten as the expression on the right-hand side.

The function **MATCH**(*pattern,exp,env*) is used to determine whether or not an expression 'exp' matches 'pattern'. 'Env' is an association list pairing meta-variables and expressions; it is ordinarily NIL when MATCH is called at top-level.

The output of MATCH is either an association list pairing meta-variables and expressions or the atom FAIL. Thus:

```
(MATCH (ELEMENT-OF FREDERICK (SETOF VINSON))
       (ELEMENT-OF $x (SETOF $a))
       nil)
```

evaluates to the association list '(($a . VINSON)($x . FREDERICK))

This output is then given to the function which handles the right-hand side of a global transformation. This function is **META-EVALUATE**(*exp,env*). Its effect is take an expression containing meta-variables, and an environment which assigns those meta-variables, and return the resulting instantiation. As an example:

```
(META-EVALUATE '(EQUAL $a $x)
               '(($a . VINSON)($x . FREDERICK)))
```

returns the expression '(EQUAL VINSON FREDERICK)'.

META-EVALUATE is used in the initial semantic interpretation to EFL, where the semantic rules are like the right-hand-sides of global transformations.

A function DEFTRANSFORMATIONSET is provided for defininng arbitary sets of global transformations. The function APPLY-TRANSFORMATION-SET applies such a set of transformations, working in a recursive descent fashion. The function APPLY-TRANSFORMATIONS-REPEATEDLY applies a given set of transformations over and over again, until no more can be applied. It is the basis for the function SIMPLIFY.

3.1.3.6 Functions for Comparing Types

There are two major functions for comparing types. These are SUB-TYPE?, which computes the subsumption relation between types, and TYPE-INTERSECTION, which takes two type expressions and returns a third which is the "largest" sub-type of both.

For any two types t1, t2, and for all indices of evaluation,the following statement is true of SUB-TYPE?

```
SUB-TYPE?(t1,t2) → DEN(t1) ⊆ DEN(t2)
```

This simply says that if t1 is a sub-type of t2, it is always the case that the denotation (i.e. the domain) of t1 is a subset of the denotation of t2. Note that the converse is *not* true: that is, if the denotations of one type expression is a subset of another, even at all indices of evaluation, it is not necessarily the case that the type of the first expression is sub-type of the type of the second expression.

The algorithm which computes SUB-TYPE? is now presented. In order to deal with union types we will first need the function COMPONENTS, which takes a type expression and returns the set of basic type expressions that make it up:

```
COMPONENTS(type) =def

    if   type = NULL-SET
    then {}
    else if BC(type) = UNION
            then compute-union(for: t
                               in:  type.sets
                               take: COMPONENTS(t))
            else {type}
```

where 'compute-union' takes a set of sets and returns the set that is the union of these sets. Note that the distinguished type NULL-SET has an empty set of components as is appropriate for it.

The following are example results of COMPONENTS when applied to various type expressions:

```
COMPONENTS(DESTROYERS) → DESTROYERS
COMPONENTS(UNION(A,B,C)) → {A,B,C}
COMPONENTS(UNION(A,UNION(B,B),A) → {A,B}
COMPONENTS(FUN(A,B)) → {A,B}
```

We can now present the algorithm for SUB-TYPE?:

```
SUB-TYPE?(T1,T2) =def

   if T1 = NULL-SET
   then TRUE
   else if T2 = NULL-SET
           then FALSE
           else for: x
                    in: COMPONENTS(T1)
                    holds: for: y
                               in: COMPONENTS(T2)
                               exists: COMPONENT-SUB-TYPE?(x,y)
```

where COMPONENT-SUB-TYPE? is defined by:

```
COMPONENT-SUB-TYPE?(T1,T2) =def

    BC1 <- BC(T1)
    BC2 <- BC(T2)
    if BC1 = CONSTANT and BC2 = CONSTANT
    then T1 = T2
    else if BC1 = BC2 and BC1 ε {L,B,SETS,F}
            then SUB-TYPE?(T1.ELEMENT-TYPE,T2.ELEMENT-TYPE)
            else if BC1 = BC2 and BC = TUPLES
                    then for: x
                              in: T1.ORDERED-ELEMENT-TYPES
                              as: y
                              in: T2.ORDERED-ELEMENT-TYPES
                              holds: SUB-TYPE?(X,Y)
```

3.1.3.7 Syntax Checkers for Logic Expressions

Obviously it is not convenient for humans to use the constructor functions to write down logic expressions. For this reason, an external form of the logic is provided for their use. This external form is a LISP S-expression, which the function PARSE-EXP turns into an internal form expression, checking for errors or omissions as it does so. If errors are found appropriate messages are printed on the user's terminal and the function returns NIL to its callers.

Errors include improper syntax for branch-categories, use of undeclared symbols, and use of forbidden symbols (such as branch-category names) as terms. Both regular and meta-variable expressions are parsed.

Since the conversion from external to internal form is always required, PARSE-EXP is invoked at every knowledge-base entry point. Any transformation, translation or rule that does not pass PARSE-EXP is simply refused, and the user so notified. In this way the system is protected from a great many errors that would otherwise only appear at run-time.

Another function is provided to check whether the type of expressions is meaningful. Called CHECK-EXP-TYPE, it prints out any anomalous constituent.

Finally, it is often necessary to turn internal form back into external form for user readibility. The function PPL does the conversion and the function SHOW-EXP does conversion and pretty-printing.

3.2 Structural Semantics

Structual semantics is that level of semantic analysis that is the direct consequence of the syntactic structure of an utterance. The first subsection presents the mechanism behind the semantic rules, and the algorithm that applies them. The remainder of the section is divided into several sub-sections, each dealing with one of the major parts of speech. First, the semantics of the noun phrase will be presented in 3.2.2. Next, the semantics of the verb phrase will be discussed in 3.2.3. A third section, 3.2.4 will present the semantics of whole clauses and sentences. Finally, Section 3.2.5 will be devoted to the semantics of adjective phrases.

3.2.1 The Mechanism of the Semantic Rules

The structural semantic rules are paired one-for-one with the rules of the grammar. An example of this pairing is the following:

```
  S → NP VP OPTSADJUNCT
 semantics
(meta-lambda ($np $vp $oa) ($oa (intension ((q $np) $vp))))
```

This is the rule generating declarative clauses (here presented in an abbreviated form), paired with its corresponding semantic rule. The semantic rule is cast in the form of a function from logical expressions to logical expressions, as is signified by the symbol "meta-lambda". Note that there are three variables bound in the lambda—$np, $vp $oa—and that these correspond to the three terms on the right-hand side of the syntactic rule—NP, VP and OPTSADJUNCT. This is true of all paired syntax/semantic rules; semantic rule always has the same number of arguments as the syntax rule has terms on its right-hand side.

During semantic interpretation, the semantic translations of these right-hand terms are substituted in for the argument variables of the semantic rule. The semantic translation is built up in a recursive descent over the parse tree bottoming out at terminal elements whether these are grammatical formatives or lexical items. In the case of grammatical formatives, the semantic translation is given by a semantic rule of zero arguments. In the case of lexical items, the semantic translation is given in the lexicon.

The next several subsections show how this machinery is applied in various areas of the grammar, begining with the semantics of the noun phrase.

3.2.2 Noun Phrase Semantics

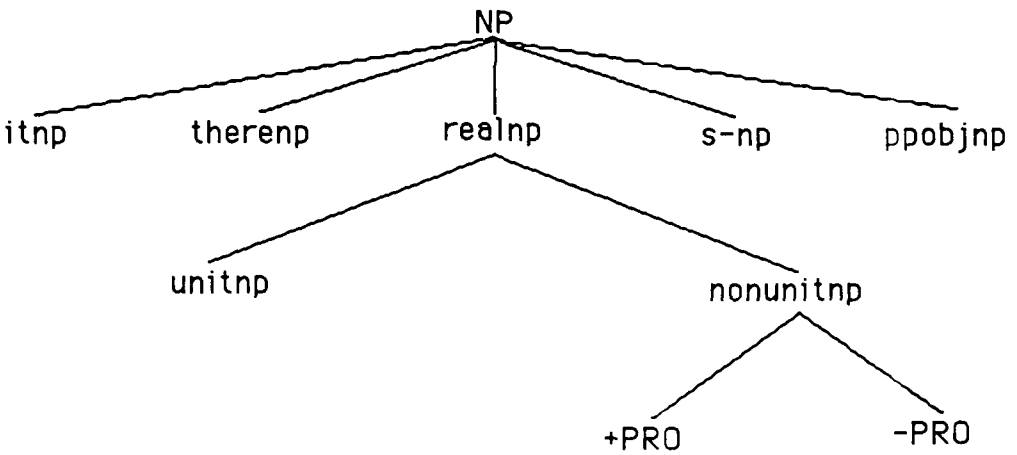


Figure 3-1: NP Types

From a syntactic point of view there are a variety of different kinds of noun phrase, as the hierarchy of noun phrases pictured in Figure 3-1 makes clear. Since these syntactic distinctions have semantic ramifications, as well, we will work our way down this hierarchy in discussing the semantics of the noun phrase. We will postpone discussion of the more unusual NP types at the top of the hierarchy—ITNP, THERENP, S-NP and PPOBJNP— so that we may first discuss the more common "REALNPs".

3.2.2.1 "Real" Noun Phrases and their Parts

The first (and most common) kind of "real" NP we will discuss are "non-unit" NPs: those whose heads are nouns like "ship" instead of units of measure like "centimeter". The semantics of UNIT-NPs will be discussed later, in 3.2.2.7.

Non-unit noun phrases whose heads are ordinary count nouns are generated by the following rule:

```
NP → DETERMINER OPTNONPOSADJP OPTADJP N-BAR OPTNPADJUNCT
semantics
(meta-lambda ($det $ona $oa $nb $on)
        ($det ($ona (set x $nb (and ($oa x) ($on x))))))
```

Note that the N-BAR serves as the range of quantification of the set expression (and is itself translated as a set). The OPTADJP and OPTNPADJUNCT serve as filters that further restrict this range. The OPTNONPOSADJP is a function taking for its argument the entire set so derived and producing another (possibly smaller) set. Finally, the DETERMINER is also a set-taking function; it returns the value that is the semantics of the entire noun phrase.

Discussion of this rule will take up the bulk of this section on noun phrases. We start by discussing the semantics of each of its parts in turn, beginning with the N-BAR.

3.2.2.2 The N-BAR

There are three ways of producing an N-BAR which we will discuss. First, an N-BAR can be produced directly from head nouns, such as "ship" or "ocean", which do not require complements:

```
N-BAR → N
semantics
(meta-lambda ($n) $n)
```

Nouns of this type are assigned sets as their semantic translations. As can be seen from the rule, this allows the translation of the noun to be passed up as the translation of the whole N-BAR. That is, N-> "ships" will given the semantic interpretation of "ships" to the N and N-BAR will have the same semantic value.

N-BARs can also be produced through combination of the head noun and one or more complements that it specifies. Examples are "list *of* spies" (one complement), "arrival *of* the ship *in* port" (two complements). Below is the rule for a noun taking a single complement:

```
N-BAR → N CASE-PHRASE
semantics
(meta-lambda ($n $cp) ($n $cp))
```

Nouns which can enter into this rule are subcategorized to take a single NP as an argument. Such nouns are translated as functions taking sets as arguments and producing sets as values. The semantic rule assumes that the translation of the CASE-PHRASE is a set (a more complete description of the semantics of case-phrases is given in the section on Verb Phrase semantics, 3.2.3).

Other argument-combining rules for N-BAR involve different numbers and types of complement phrases. Nouns in these subcategorizations are translated as set-valued functions taking corresponding numbers and types of arguments.

Complement-taking nouns are frequently called "relational", because they are associated with a relation between two or more objects (recall that a set-valued function is just another way of representing a relation). As will be discussed in sub-section 3.3.3 ("Relational Nouns") our work takes the view that some relational nouns can get their semantic arguments from NP adjuncts instead of from complements (compare "the speed of Frederick"/"the speed that Frederick has" vs. "the list of the spies"/"the list the spies have").

A third way to produce an N-BAR is through noun-noun compounding, as in such phrases as "equipment readiness" or "mission area":

```
N-BAR → N N
semantics
(meta-lambda ($n1 $n2) (set x $n2 (premod x (intension $n1))))
```

This semantic rule constructs a subset of the head noun translation "n2" by stipulating that its elements be related to

the intension of "n1" by the relation "PREMOD". This is an EFL constant used as a placeholder at the level of structural semantics; its actual value is determined by the lexical semantics component (EFL to WML translation).

Some comments on this rule seem appropriate. PREMOD is combined with the intension of the set "n1" and not the set "n1" simply because nominal compounding is not an extensional phenonenon. Consider the term "fire alarm": what a "fire alarm" is is not describable with reference to any one fire, nor even with reference to all fires going on *at this moment*. Instead, it is related (somehow) to the *concept* of fires in general and the intension is the closest we can get to this within the formal apparatus of intensional logic.[27]

In line with this, we should point out a problem with the rule: only the second argument-place of PREMOD is intensional. This would rule out possible compounds in which the extension of the whole is not a subset of the extension of the head noun. (For a possible example, consider the compound "water cannon": this is surely not a "cannon" in the normal sense of field artillery.) The obvious solution is to make PREMOD combine with the intensions of both "n1" and "n2".

3.2.2.3 The OPTADJP

The OPTADJP modifies the N-BAR. The OPTADJP generates a string of zero or more positive adjective phrases (those whose heads are not inflected as comparative or superlative). OPTADJPs are always translated by one-place predicates. The syntactic and semantic rules are those below:

```
OPTADJP → ()
semantics (lambda (x) gts true)

OPTADJP → ADJP OPTADJP
semantics
  (meta-lambda ($a $oa) (lambda (x) gts (and ($a x) ($oa x))))
```

where the type "gts" embraces individual entities and amounts, as well as sets (and sets of sets, etc.) of these.

In the first rule, there are no adjective phrases and the OPTADJP generates the empty string. Accordingly, the predicate produced as its translation returns true for any (allowable) argument. In the second rule there is at least one adjective phrase. The rule constructs a new predicate which returns true for an argument only if it satisfies both the translation of the ADJP and the tr nslation of the recursive OPTADJP. In this way, any number of adjective phrases can be accomodated as restrictions on the translation of the N-BAR.

The limitation of the second rule is that it is only adequate for handling so-called "intersective" adjectives: those which can be semantically translated as a property that can be "intersected" with the set translating the noun to give a set that is the whole. An example of an intersective adjective is "red": a "red ball" is something that is both a "ball" and which has the property "red".

---

[27] One general solution to this that some workers, e.g. Carlson [8], have proposed is to have a notion of "kinds" as actual individuals in one's ontology. A version of this has not yet been implemented in our system.

Unfortunately not all adjectives fall into this category. Some adjectives, like "big" and "good", have a meaning that depends on the noun they are combined with. A "big elephant" is not the same size as a "big mouse": there is no fixed property of big-ness independent of the set of individuals being looked at. An adjective like "big" would have to be translated as a set-taking, set-valued function, which the rule above does not accomodate.

For an adjective like "good" a simple extensional set-taking function is not enough. Consider an index of evaluation in which the teachers happen to be the same as the coaches. A "good teacher" at this index is not the same as a "good coach"; we have only to consider an individual, say "Mr. Smith", who is an excellent coach but an abominable teacher. Such an adjective would have to be translated as a function taking the intension of a set-expression and returning an extensional set.

The same is true for certain other adjectives such as "alleged" and "former". Note that these adjectives are not even "restrictive" as "good" and "big" were: though all "good teachers" are teachers, "former Communist" are, by definition, not Communists. The adjective "former" can be readily treated as a function which takes the *intension* of the set expression translating "Communist" and delivers those individuals who were members of the set at indices prior to the current index but who are not members now. Once again, however, this cannot be accomodated by the rule above.

Finally, we have adjectives like "previous" and "average" which seem to prefer complement-taking nouns: "previous readiness", "average speed". These adjectives are also sometimes intensional in character: the "previous readiness" of a ship is the value it had before the value it has now, which requires the intension of "readiness".

These different types of adjectives are summarized in Figure 3-2.

There are at least three possible solutions to these problems. The first is to assimilate all adjectives to the worst case (intensional-set) and provide meaning-postulates for the other cases that would "lower" them to an extensional or intersective level. (This leaves open the issue of what to do when an adjective phrase is made into a VP—e.g. "is red".) The second is to mark the different semantic kinds of adjectives with some syntactic diacritic, and have corresponding syntactic/semantic rules that combine adjectives one at a time with the head noun instead of assembling them into an OPTADJP (much like categorial grammar). Finally the third alternative, intermediate between the first two, would be to have a separate slot in the NP rule for non-intersective adjectives that precedes OPTADJP, on the belief that there are independent grounds for this (compare "big brown dog" with "brown big dog"). Which path to take is a question for further work.

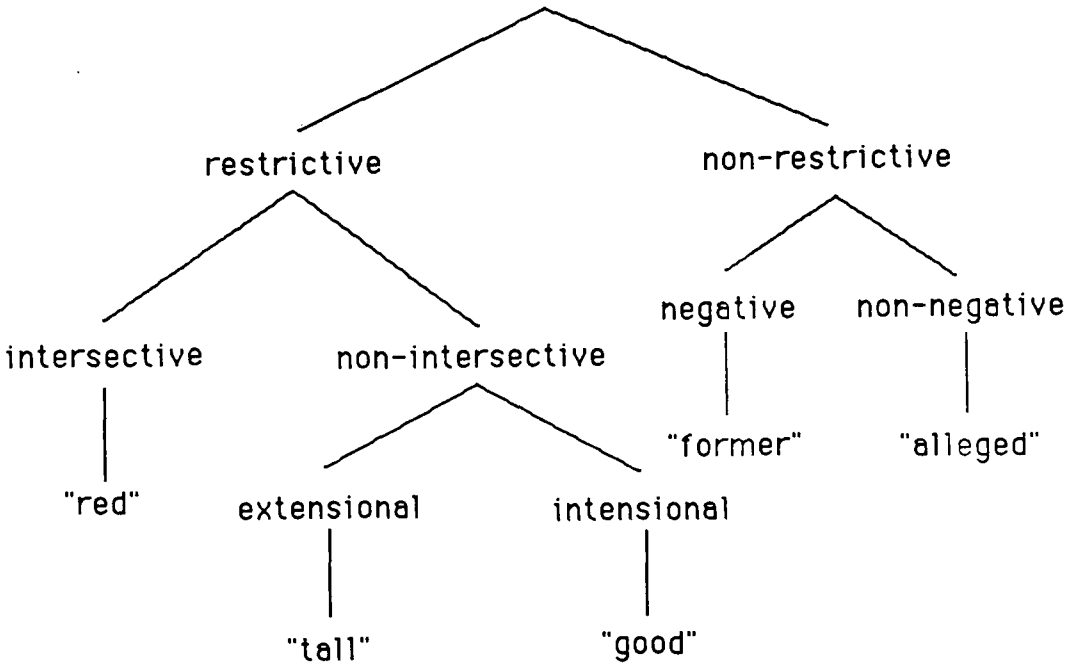


Figure 3-2: Types of Adjectives

3.2.2.4 OPTNPADJUNCTs and Relative Clauses

OPTNPADJUNCTs also modify the N-BAR. OPTNPADJUNCTs (in keeping with their name) can be empty, or they can be prepositional phrases of various kinds (''in the Indian Ocean''), and verb phrases (''heading towards the indian ocean''). Although the syntactic possiblities for OPTNPADJUNCT are more complicated, the semantic mechanisms are similar to those for OPTADJP.

Relative clauses (''that are in the Indian Ocean'') are given the same one-place predicate semantic treatment, but are combined into an NP via a separate syntactic rule.

3.2.2.5 The OPTNONPOSADJP

The OPTNONPOSADJP is the slot for optional superlative/comparative modification of a noun: ''The fastest, biggest ship'', ''The bigger ship''. We will discuss only the superlative modification of nouns, since comparative modification of nouns was not implemented. (3.3.2 gives a treatment for predicative instances of the comparative, such as ''Frederick is bigger than Vinson'')

A noun phrase like ''the fastest ship'' requires that the translation of ''fastest'' look at the whole set of ships collectively in order to find the fastest one. Accordingly, the OPTNONPOSADJP translation takes a set as its arguments (the translation of the N-BAR with restrictions imposed by OPTADJP and OPTNPADJUNCT) and delivers a subset of this set as its value. Following are the two rules building OPTNONPOSADJP:

```
OPTNONPOSADJP →
semantics (lambda (s) (sets gts) s)

OPTNONPOSADJP → ADJP(non-positive) OPTNONPOSADJP
semantics
(meta-lambda ($a $o)
    (lambda (s) (sets gts)
        (intersection ($o s)
                      (set x s (((lambda (svar) gts
                                    ((lambda (sup-num) integers $a) 1))
                                 s)
                                x)))))
```

In the first rule, the OPTNONPOSADJP is empty and its translation simply returns the set it is given. This would be the path taken in the interpretation of a noun phrase "the red ship" which has no superlative.

The second rule handles one or more superlatives ("the fastest, tallest ship"). Multiple superlatives are handled by intersecting the results of two set-taking functions—the translation of the recursive OPTNONPOSADJP and a set-taking function constructed from the translation of the superlative ADJP. Thus, if the tallest ship isn't also the fastest, the empty set will be returned.

As discussed in Section 3.2.5, the translation of a superlative ADJP is a one-place predicate which picks out the individual or individuals that fit the superlative ("Frederick is *fastest*"). The translation of the ADJP "fastest" is:

```
(lambda (x) gts
   (element-of x (first supnum
                        (sort supvar
                              (lambda (x y) (supvar supvar)
                                 (larger-than (speed-of x)
                                              (speed-of y)))))))
```

This predicate expression has two free variables which the OPTNONPOSADJP rule fills in. The first free variable, "s-var", is the set over which superlativization is to be performed. This is filled in by the set that the OPTNONPOSADJP is to be applied to. The second free variable, "sup-num", is the distance to go in the rank-ordering of the superlative ("the fastest", "the two fastest", "the three fastest" etc.). It is currently supplied with the default value of 1 (this is only adequate for singular NPs—compare "the fastest ship" with "the fastest ships").

Constructions with a numeric or ordinal applied to the superlative, such as "The three fastest ships" or "The second fastest ship", are handled by separate rules. The section on adjective phrases (3.2.5) gives more detail on the handling of the superlative.

3.2.2.6 The DETERMINER

Finally, the translation of the DETERMINER is applied to the combination of the N-BAR, OPTNONPOSADJP, OPTADJP and OPTNPADJUNCT to give the translation of the entire NP. The translation of the determiner is a set-taking function. The value returned by this function depends on the specific determiner in question.

Definite Determiners

Definite noun phrases are always translated as terms without quantifiers. There are two rules by which DETERMINER can go to "the" for count nouns, corresponding to singular and plural forms of the noun. In the singular form, the translation is:

```
(meta-lambda ($n) (sets gts) (setof (the $n)))
```

The "THE" operator in this rule is a discourse-selection operator: applied to a set it picks out the element of the set which is salient in context. That this contextual saliency is not the same as a presupposition of definiteness can be seen from an NP like "the ship". A speaker using this NP as a description is unlikely to believe that there is just one ship in the world. Of course, a more complex description may have a single satisfier, in which the "THE" operator delivers just that one element. Because the translation type for (unquantified) NPs is sets, the operator "setof" is neccesary to turn that element into a singleton set.

When "the" is to be combined with a plural noun it is generated by a different syntactic rule[28]. Its translation is:

```
(meta-lambda ($n) (sets gts) (setof (the-plural $n)))
```

The only difference between these two rules is that the "THE" operator has been replaced by a "THE-PLURAL" operator which returns a contextually salient *subset* of its argument, instead of a single element of the argument.

Note that the plural definite in our system is *not* treated as a universal quantifier (as might be the choice in many other systems) but is instead treated as a set-valued term. This has the consequence that predication made of a plural definite noun phrase is *collective*: it is made on an entire set at once instead of each of its elements separately. Further discussion on this point will be found in Section 3.3.5.

Very similar semantic rules and operators are provided to handle the determiners "this"/"these" and "that"/"those". The same remarks about plurality and collectiveness apply to them.

Quantificational Determiners

When DETERMINER generates "every" or "each" (and is combined with a count noun) its semantic translation is a more complex "generalized quantifier" ala Montague's PTQ [35]:

---

[28]This is accomplished by splitting the rule DETHEAD-—"the" into two rules based on the value of **number**. Such an approach is necessary because our semantic rules do not currently have the ability to look at syntactic features.

```
(lambda (n) (sets gts) (lambda (p) preds (forall x n (p x))))
```

This takes a set and delivers a rather unusual predicate: one taking predicates themselves as arguments. The translation of "every ship" would be, after lambda-reduction:

```
(lambda (p) preds (forall x SHIPS (p x)))
```

This is a predicate on predicates (of a single individual) which returns true if its argument returns true for all ships. The translation for "no" as a determiner is a similar function, exce . that it inverts this stipulation:

```
(lambda (n) (sets gts) (lambda (p) preds (not (exists x n (p x)))))
```

Generalized quantifiers are also used in the translations of such determiners as "three":

```
(lambda (n) (sets gts)
        (lambda (p) preds (equal (cardinality (set x n (p x))) 3)))
```

and "more than three":

```
(lambda (n) (sets gts)
        (lambda (p) preds (greater (cardinality (set x n (p x))) 3)))
```

Note that quantifiers in our work are interpreted "in situ": we did not address the issue of quantifier scope. Later work may incorporate a separate quantifier ordering pass, or take a different approach to quantifiers.

The translation for "a" and "some" as count determiners does not directly generate such predicate-functions. Instead, the entire set is passed along unmolested:

```
(lambda (n) (sets gts) n)
```

This is raised to a generalized quantifier when the NP is combined with a verb (see the section on VP semantics, 3.2.3).

Possessive Determiners

Examples of possessive determiners are "*Frederick's* speed", "*Every man's* mother", etc. These determiners are generated from possesive NPs by the rule:

```
DETHEAD → NP(poss-np)
semantics
(meta-lambda ($np)
        (lambda (n) (sets gts)
                (setof (the (set x n ((q $np) (lambda y gts (have y x))))))))
```

The "q" operator above assimilates all nps to generalized quantifier level and is described in the section on VP semantics (3.2.3). It is included so the rule can combine not only with quantificational NPs like "every man", "three ships", etc. but with non-quantificational NPs (proper, definite, indefinite) as well.

When applied to an NP, say "John", this rule gives the following translation for the possessive determiner (when it is to be combined with the singular form of the noun):

```
(lambda (n) (sets gts) (setof (the (set x n (have John x)))))
```

This is a function from sets "n" (the set translating the rest of the NP) to singleton sets whose element is the contextually salient element of the subset of "n" consisting of things "had" by John. Note that this contextual

salience is enforced by the same "THE" operator that was seen in the translation of the definite determiner above. The "THE" operator is necessary because a singular possessed NP like "John's car" has an implicit definiteness: John might have many cars, but we are only talking about the one in context.

When a possessive determiner is to be combined with a plural head noun the "THE" operator is replaced by "THE-PLURAL" (just as in plural definite NPs).

Before leaving the topic of possessive determiners, it is necessary to point out a problem in the interaction of possessive determiners with superlative OPTNONPOSADJPs. Consider the NP "John's best idea". Let us skip over the details of how the superlative "best" is handled, and render it simply as the set-taking function "BEST". Then the NP will be translated as:

```
(setof (the (set x (best ideas) (have x John))))
```

This translation extracts from the set of "best ideas" those which were had by John, and then selects the contextually salient single element of this set. But this not correct: the elements of the set of "best ideas" are those which were had by all sorts of people, perhaps including, but certainly not limited to, John. What we really want is:

```
(setof (the (best (set x ideas (have x John)))))
```

but this is impossible given the way determiners and superlatives are to be structurally combined in our rule for NP.

This is essentially a matter of scope. One way to handle it (a technique used by the RUS system [3]) is to treat the possessive determiner as a complement or adjunct by effectively transforming "John's best idea" into "The best idea of John". This would have the desired effect: the modifier "of John" would then be able constrain input to the superlative function "best" instead of its result. It is not obvious, however, how we would implement this solution in our grammar and parsing framework. Moreover, it is not even clear that this treatment is correct in the general case. Consider the NP "The spies' list", which has to mean the list (of whatever) that the spies have in their possession. Certainly this is *not* equivalent to "The list of the spies", which can only mean the list whose contents are the names of the spies[29].

We are currently considering changes to the syntax which would assimilate the superlative to the determiner structure itself, and thus allow them to be interpreted together. This is more generally motivated by the tendency of superlatives to mingle with what would otherwise be regarded as determiner elements ("the three fastest"/ "the fastest three") and to make other determiner elements semantically superflous (the "the" in "the fastest ship").

NPs without determiners

This section is concerned with NPs that do not have determiners: bare plurals, bare mass nouns and the more unusual category of bare singulars. These are not generated by the rule for NPs presented earlier, but by their own, determinerless rules.

[29]One could of course add a fictional "preposition" which could be distinguished from "of", but this does not seem very attractive

Bare plural NPs are translated as sets:

```
NP → OPTNONPOSADJP OTPADJP N-BAR OPTNPADJUNCT
semantics
(meta-lambda ($ona $oa $nb on) ($ona (set x $nb (and ($oa x) (on x)))))
```

Thus, the bare plural "ships" is translated simply as "SHIPS". Since this is the same as the translation for "some ships" our system treats them as one and the same. This will turn out to mean (as we will see in the section on clausal semantics) that the system treats "John sells *cars*" in the same way as "John sells *some cars*" and "*Children* make noise" in the same way as "*Some children* make noise".

Bare mass nouns are generated by the following rule:

```
NP → OPTNONPOSADJP OPTADJP N-BAR
semantics
(meta-lambda ($ona $oa nb) (setof ($ona (set x $nb ($oa x)))))
```

This rule means that a bare mass noun like "equipment" will receive the semantic translation

```
(SETOF EQUIPMENT)
```

This turns out to mean that predication made on a mass noun is always treated collectively.

A very few count nouns are able to appear in a bare singular form, much like a mass noun. Examples from our domain are words like "ntds" in "Frederick has *ntds*" instead of "Frederick has *an ntds*". Bare singulars are treated semantically in the same way as bare mass nouns: predication made on them is collective.

WH Determiners

Our last category of determiners are the WH determiners. These also have a complex type, as may be seen in the translation for "what" and "which" (when combined with count nouns):

```
(lambda (n) (sets gts) (lambda (p) preds (lambda (x) n (p x))))
```

When combined with the other elements of the noun phrase, as in "what ships", this function yields the following value for the complete NP:

```
(lambda (p) preds (lambda (x) SHIPS (p x)))
```

This is a function mapping the predicate "p" to a "subset" of "p" whose domain is restricted to be the set "SHIPS".

The translation for "how many" as a determiner is as follows:

```
(lambda (n) (sets gts)
   (lambda (p) preds
      (lambda (x) integers (equal x (cardinality (set x n (p x)))))))
```

When combined with other elements of the noun phrase, as in "how many ships", this function yields the following value for the complete NP:

```
(lambda (p) preds
   (lambda (x) integers (equal x (cardinality (set x SHIPS (p x))))))
```

This is a function mapping predicates on individuals "p" to predicates on integers which return true if the integer is equal to the the number of ships that satisfy the predicate "p".

Finally there is the possessive WH determiner "whose":

```
(lambda (n) (sets gts)
   (lambda (P) preds
      (lambda (x) agents (P (the (set y N (have x y)))))))
```

When combined with the rest of an NP, as in "whose ship", this forms:

```
(lambda (P)     .ds
   (lambda (x) agents (P (the (set y SHIPS (have x y))))))
```

This is a function from predicates "P" to predicates on agents such that the predicate "P" is true of the contextually salient ship possessed by the agent.

How these WH-NPs combine with a VP to form a WH sentence is described in the section on clausal semantics 3.2.4.

3.2.2.7 Unit NPs

Unit NPs are those which refer to a measurement and not an object or set of objects. As an example, consider the following:

"The distance from Frederick to Vinson is *five miles*"

Clearly in this example, the noun phrase *five miles* is not to be construed as some set of five "mile entities" that is (somehow) identical to the distance from Frederick to Vinson. Rather, it indicates a measurement.

In our system, this is handled by the notion of an "amount" [47]. An amount is pair consisting of a number and a "unit of measure" (which is actually itself a primitive amount). Thus the semantics of "five miles" (in its ordinary sense) is represented as:

```
(amount 5 miles)
```

Unit NPs are separated from non-unit NPs on syntactic grounds as well (see Section 2.5.6 of the syntax chapter). Since completely different sets of rules produce unit and non-unit NPs their different kinds of semantics do not interfere with one another.[30]

3.2.2.8 NPs which are not "real" NPs

There are several small (or unit) classes of NP which are not "real" NPs but which are necessary on syntactic grounds. First, there is the "THERENP", as in "*There* is a ship in IO". This is a grammatical formative, and is given the semantic translation "THINGS"

[30]There is a "coercion" rule that lifts unit nouns to non-unit nouns, in order to handle utterances like "That last mile was a killer", where it is necessary to regard the NP as an individual.

The ITNP, as in "*It* is raining" is translated as

```
(SETOF DUMMY)
```

This is simplified away in later processing.

The S-NP is the category of nomininalized sentence, as in "*That John left* surprised me". We currently do not provide semantics for such NPs.

Finally, there are the PPOBJNPs, generated from those nouns which can appear directly as the object of a preposition, as in "Frederick is in *port*". These are assigned the set semantics of their head noun, and therefore the remarks for bare plural nps apply here. The sentence "Frederick is in port" is treated as:

```
(exists x PORT (in x Frederick))
```

3.2.3 Semantics of the Verb Phrase

Verb phrases are translated into one-place predicates, which are then applied to the subject of the clause. A verb phrase is built up out of a verb and zero or more complement and adjunct phrases.

3.2.3.1 Simple VPs

The simplest VP is just that which results in an intransitive verb like "walk" or "sail":

```
VP → (V (intransitive))
semantics (meta-lambda ($x) $x))
```

Intransitive verbs are translated into one-place predicates. Note that this semantic rule simply passes this verb translation up as the translation of the entire VP.

Verb Phrases can also be built out of either a noun phrase, adjective or prepositional phrase using the auxillary verb "be". Examples are "Frederick is c3" and "Frederick is in the Indian Ocean". The rule generating such VPs is:

```
VP → (V (be)) VP
semantics (meta-lambda ($v $vp) $vp)
```

the semantics of the internal VP is again passed up as the semantics of the whole.

Both adjectives and prepositional phrases are translated by one-place predicates, and their translations are passed upwards as the translation of the VP in exactly the same fashion as intransitive verbs. The rule for a predicate nominal VP—such as the VP in the derivation of "Frederick is a ship"—is as follows:

```
VP → NP
semantics
(meta-lambda ($np) (lambda (x) gts (BE x $np)))
```

The meaning of the EFL constant "BE" is worked out in the second, lexical stage of semantic processing.

3.2.3.2 Transitive VPs

A more complex case is that of transitive verbs, as is the verb "deploy" in "Pacflt deployed Frederick". The rule generating the VP "deployed Frederick" is given below.

```
VP→ (V (transitive (takes-active)) NP
semantics
(meta-lambda ($v $np) (lambda (x) gts ((Q $np) (lambda (y) gts ($v x y))))))
```

Note the use of the "Q" operator to supply the second argument to the verb.

The 'np' element can be one of two things: either a set of individuals (which may have just one member) or a rather unusual kind of function: one taking predicates and delivering truth-values; that is, a predicate on predicates. As was explained in the section on determiners, 3.2.2.6, this predicate has the effect of picking out of all predicates just those which are true of the translation of the NP. By way of example, consider the following such predicate:

```
(lambda (P) (forall x SHIPS (P x)))
```

This has the effect of picking out of all predicates just those which are true of every ship. It can be compared with the similar-looking predicate expression:

```
(lambda (P) (exists x SHIPS (P x)))
```

which picks out all predicates true of at least one ship, and:

```
(lambda (P) predicates (P Frederick))
```

which picks out all predicates true of the ship "Frederick". All of these complex predicate expressions are termed in the literature "generalized quantifiers".

The "Q" operator in the semantic rule is applied to the translation of the NP and is essentially a coercion operator. If it receives as argument such a "generalized quantifier" it passes it along undisturbed. If on the other hand it receives a set of individuals as the translation of the NP it lifts it to a generalized quantifier using the following functional schema:

```
(lambda (S) (lambda (P) (exists x S (P x))))
```

When the combination "(Q np)", producing such a function, is applied to the Verb Phrase translation "vp", a truth-valued formula results. For example, the result of applying (Q np) to vp in the case where np translates "every ship" and the vp translates "sails" would be:

```
(forall x SHIPS (SAIL x))
```

The result for "Frederick sails" would be:

```
(exists x (setof Frederick) (SAIL x))
```

which of course is directly reducible to `(SAIL Frederick)`.

Passivization, with and without a by-phrase, is handled in the following rule:

```
VP → (V (transitive (takes-passive)) OPTBY
semantics
(lambda ($v $bp) (lambda (y) gts ((Q $bp) (lambda (x) gts ($v x y))))))
```

Note that the semantic rule here is structurally identical to the active form, with the translation of the OPTBY, "bp", playing the same role as the translation of the object NP. The OPTBY can generate either a phrase "by <NP>" or the empty string. In the former case, the translation of the NP is passed up as the translation of the OPTBY. In the latter case, governed by the rule:

```
OPTBY →
semantics
(lambda () (set x gts true)))
```

the set of all individuals is passed up. When the Q is applied to this set, an existential quantification results. Thus the translation of "Frederick was deployed" is:

```
(exists x gts (deploy x Frederick))
```

3.2.3.3 Other sub-categorization Frames

Another sub-categorization frame is that for dative verbs—those taking a subject, object and indirect object introduced by a preposition. An example of use of a ditransitive verb is "give" in "John gave the book to Mary" The rule generating the VP "gave the book to Mary" is:

```
VP → V NP CASE-PHRASE
semantics
(lambda ($v $np $cp)
   (lambda x gts ((q $np) (lambda y gts
                            ((Q $cp) (lambda z gts ($v x y z)))))))
```

The NP in this example is of course "the book", and the CASE PHRASE "to Mary". CASE-PHRASEs in our system, while superficially similar to prepositional phrases, are syntactically and semantically distinct from them. They serve to mark arguments to a verb (or other categories, such as ADJ). Their semantic translations are simply the semantic translations of their object NPs: the preposition-like element "to" does not contribute to their semantics. It is useful here to think of the "to" as a kind of argument label or keyword.

Note from the form of the rule that a ditransitive verb must be translated by a three-place predicate ("v") to which the translations "np" and "cp" are conveyed as arguments by the Q operator.

Particles, such as the word "up" in the sentence "They blew up the ship" or "They blew the ship up", have a similar non-contributory status. We take the position that they are part of a single discontinous constituent, and thus "up" does not have a separate semantics to contribute here. The syntactic rules generating these two sentences have identical semantic rules (except for argument order) corresponding to them. The first is:

```
VP → (V (TRANSPART)) PARTICLE NP
semantics
(lambda ($v $p $np) (lambda x gts ((q $np) (lambda y gts ($v x y)))))
```

Note that verbs in the TRANSPART subcategorization frame are translated by two-place predicates.

Still another kind of verb sub-categorization takes an adjective phrase instead of a noun phrase as a complement. An example is "(Frederick) became C3", where "C3" is an adjective from our domain. Such constructions are generated by the rule:

```
VP → V ADJP
semantics
(lambda ($v $a) (lambda (x) gts ($v x (intension $a))))
```

This rule must take the intension of the ADJP translation because its extension (as a simple predicate) is not enough information. The statement "Frederick became C3" is true at a particular time *t* only if the statement "Frederick is C3" is false at a time just before *t* and true at a time just after *t*. It is not enough that Frederick *is* C3 at a particular time—a change has to have just taken place to make it that way. To get at this information, we need the intension of the predicate—a function over the indices that gives the value that the predicate has at these indices.

This issue is not peculiar to just this VP rule, but arises instead in all VP rules where a predicating phrase—an ADJP, a PP, a predicate nominal or another VP—serves as an argument to the head verb. Consider "Frederick arrived at San Diego" (loc-temp complement) and "Frederick is expected to go to San Diego" (infinitival VP complement). In neither case is the extensional predicate enough to give the proper truth-conditions of the sentence. Note also that in the second example, even the temporal aspect is insufficient—we have to consider as well some set of "expected" or "normative" possible worlds in which the Frederick goes to San Diego, and which may or may not turn out to coincide with the actual future when it arrives.

3.2.3.4 Verbs with Multiple Sub-categorization Frames

As we have seen, each sub-categorization frame is associated with a particular semantic type—that of a predicate of some arity and argument specification. These have been just a few examples—we have assigned explicit semantic type to approximately 45 verb sub-categorization frames.

Many verbs can appear in more than one subcategorization frame. An example is the verb "turn". One can say "John turned the knob" (TRANSITIVE), "The knob turned" (INTRANSITIVE), "John turned on the television" and "Mary turned John on" (TRANSPART) as well as "John turned Mary on to subcategorization" and "John's pit bull turned on him". In such a case, the verb has a different EFL semantic translation for each such sub-categorization frame in which it can appear.

Of course different semantic senses can appear within a particular subcategorization frame—consider "John turned the television on" vs. "Mary turned John on". A single EFL constant is assigned for both cases, and disambiguation is handled in the lexical semantic (EFL to WML translation) component. This is discussed in more detail in the section on lexical semantics (3.3).

3.2.3.5 Modality, Negation and Conjunction

The rule introducing a modal auxillary into a VP is:

```
VP → MODAL VP
semantics
(lambda ($m $n $vp) (lambda x gts ($m (intension ($vp x)))))
```

Modals in our system take propositions as their argument and deliver truth values. Consider the modal verb "can". It is introduced by the rule:

```
MODAL → can
semantics
(lambda (x) propositions (exists i worlds (x *t* i)))
```

This function takes a proposition and delivers true if there exists a possible world at which the proposition is true at the indexical time "*t*". Note this rule is somewhat defective in that it does not accomodate a notion of context—such as a set of possible worlds which are "accessible" from the world at which the expression is being evaluated.

Negation ("Frederick can't go to Hawaii") is handled by the rule

```
NEG → n't
semantics
(lambda (x) tv (not x))
```

where "NEG" is a constituent of separate VP rules handling negated VPs (not shown).

The semantics of Verb Phrase conjunction is quite simple. Given two VP translations vp1 and vp2, they are combined into one by the semantic rule:

```
(lambda (vp1 vp2) (lambda (x) (and (vp1 x) (vp2 x))))
```

3.2.3.6 Some Unresolved Problems

As we have seen, verb subcategorizations taking phrasal categories other than noun phrases ("arrive", etc.) involve intensionality phenomena. Some verbs taking only noun phrases also turn out to require intensionality, however. Consider the intransitive verb "increase" in "Frederick's speed increased". If "increase" were treated as any intransitive verb, this would be translated as:

```
(increase (speed-of Frederick))
```

This is not correct, however. Suppose the speed of Frederick is 30 knots. Then the formula can equivalently written as:

```
(increase (amount 30 knots))
```

which is nonsense—an amount is a value simply and cannot "increase".

The correct translation of the sentence is

```
(increase (intension (speed-of Frederick)))
```

where the predicate "increase" can get at the time series of values of Frederick's speed, in order to say if (at the given index of evaluation) it increased.

The difficulty here is the requirement that the treatment for a particular sub-categorization be uniform. Our current logical framework cannot allow the verb itself to specify whether it takes intensional or extensional arguments, because substitution is not allowed into intensional contexts. That is, one could not make use of a lexical entry like:

```
''INCREASE'' => (lambda (x) (INCREASE (INTENSION x)))
```

As a "hack" to work around this problem, we have changed the definition of substitution to allow such operations, but this is not a satisfactory long-term solution.

3.2.3.7 Tense and Aspect

The treatment of tense in our system is quite rudimentary, essentially involving bound quantification over past or future times. Verb aspect was not covered in this work.

3.2.4 Clausal and Sentential Semantics

Clauses are built up out of an NP (the subject) and a VP (the predicate) together with adjuncts and a complementizer (if any).

3.2.4.1 Types of Clauses

Clauses are divided into a number of sub-classes based primarily on the value of the `compflg` feature. The clauses with a "0comp" value for this feature are non-complement clauses: both declarative and interrogative (yes-no and WH). Declarative clauses and yes/no interrogative clauses are both translated by expressions of type "TV" (i.e., as formulas). WH clauses (distinguished from their other 0comp brethren by a WH+ value of their WH feature) are translated by expressions whose type may be THINGS, SETS of THINGS, or AMOUNTS (the union type GTS).

Complement clauses have a "nonrootcomp" value of the feature complfg. These are divided into several more refined classes and are translated by predicates of various types.

3.2.4.2 Semantic Rules for the START Symbol

The START symbol is the top-node of any succesfully parsed input. It may generate a non-complement clause or an imperative VP (NPs may also serve as full utterances). The semantics for rules generating START wrap "speech-act" operators around the interpretation of the descendant clause or VP. The rules for declarative, interogative and imperative sentences are as follows:

```
START → (S (wh-) (0comp))
semantics (meta-lambda ($s) (assert $s))

START → (S (wh+) (0comp))
semantics (meta-lambda ($s) (query $s))

START → (VP (imperative))
semantics (meta-lambda ($vp)
              (bring-about (intension (lambda (x) things ($vp x)))))
```

These speech-act operators represent the performative part of the sentence (the act of questioning, asserting or commanding) and allow us to distinguish it from the propositional content of the sentence. The speech-act operator

"ASSERT" takes a proposition and stands for the act of asserting the proposition. The speech-act operator "QUERY" takes any expression at all and stands for the act of asking what the value of the expression is (as we shall see, this covers both WH and yes/no questions). Finally, the speech-act operator "BRING-ABOUT" takes a property and stands for the act of commanding the hearer to make the property true of himself.

The bulk of the clauses occupying our attention will be the "0comp" variety that can descend from the START symbol. We will first present the semantics of the declarative clause.

3.2.4.3 The basic declarative clause and its parts

Below is the combination of syntactic and semantic rules used for declarative clauses:

```
S → NP VP OPTSADJUNCT
semantics
(lambda ($np $vp $oa) ($oa (intension ((q $np) $vp))))
```

The clause constructed by this rule has three constituents: a Noun Phrase (its subject) a Verb Phrase (its predicate) and optional adjunct phrase. The semantic representation of the clause is constructed from the semantic representations of these three parts. These will now be given in detail.

The 'vp' element of this rule is a predicate, taking either individuals or collections (sets, etc.) of individuals.

The third element of the clause is the optional adjunct phrase. The translation of the optional adjunct, 'oa', is another complex function which takes propositions (the intensions of formulas) and delivers truth-values—in other words, a predicate on propositions. Applying to the intension of the formula produced above produces another formula whose truth-conditions are that the proposition is true under the conditions (or at the time or place) mentioned in the adjunct. For example, in the clause "Frederick sailed yesterday" we would have as the translation of the adjunct:

```
(LAMBDA P PROPOSITIONS (P (DAY-BEFORE NOW) *W*))
```

which when applied to the translation of the previous example "Frederick sailed" gives us:

```
((INTENSION (SAIL FREDERICK)) (DAY-BEFORE NOW) *W*)
```

This formula will be true if the formula "(SAIL FREDERICK)" was true in the time interval denoted by "(DAY-BEFORE NOW)" and in a possible world to be determined by context (the value of *W*, which normally the actual world of reality).

Of course the adjunct may be something other than a time adverbial. For "Frederick sailed using diesel" we would have:

```
(WHILE (INTENSION (SAIL FREDERICK)) (LAMBDA x (USE x DIESEL)))
```

the function 'WHILE' takes a proposition and a a predicate.

Finally, the adjunct, since it is optional, may generate the empty string. The translation of an empty adjunct is simply:

```
(LAMBDA P PROPOSITIONS (EXTENSION P))
```

The 'EXTENSION' operator cancels out the 'INTENSION' operator, leaving just a formula.

3.2.4.4 Question Clauses

Question clauses encompass both WH and yes-no questions. The semantic translation of question clauses in our system is just the expression whose value is the answer to the question.

The first kind of WH question has the WH in subject position, as in "*Who* saw Mary", "*What ships* are in IO". Such clauses are generated by the following rule:

```
S → NP(wh+) VP OPTSADJUNCT
semantics
(lambda ($np $vp $oa)
        (pred-to-set ($np (lambda x
                            ($oa (intension ($vp x)))))))
```

Recall from 3.2.2.6 that we translate WH NPs as functions from predicates to domain-restriction on the predicate. That is, "what ships" is rendered as:

```
(lambda (p) preds (lambda (x) ships (p x)))
```

This function is given as argument the translation VP of the clause (modified by the optsadjunct, if any). In the case of our example question, "What ships are in the IO?", this VP translation is:

```
(lambda (x) gts (in x IO))
```

The action of the function translating the WH NP "What ships" is to return a restricted from of this predicate where the domain is restricted to "ships". The restricted predicate resulting is:

```
(lambda (x) ships (in x IO))
```

The "PRED-TO-SET" operator in the rule takes a predicate and returns the subset of the predicate's domain for which the predicate returns true. The final result of the interpretation of the clause is thus:

```
(set x ships (in x io))
```

which is just the answer we want.

WH questions involving a trace (non-subject WH questions) and yes-no questions are generated by the same rule:

```
S → OPTWH V NP OPTPREVP VP OPTSADJUNCT
semantics
(meta-lambda ($oa $v $np $pr $vp $oa)
   ($ow (lambda trvar ($oa (intension ((q $np) $vp))))))
```

When a trace appears anywhere in the clause, its translation is "TRVAR". Abstraction over this TRVAR constructs a predicate which is passed to the OPTWH translation, "ow".

If there is no trace in the clause, the OPTWH is empty and we have a yes-no question, such as "Is Frederick in the Indian Ocean?" The empty OPTWH is generated by the following rule:

```
OPTWH →
semantics
(lambda (p) preds (p dontcare))
```

This translation is a predicate on predicates "p," returning true if the predicate is true of the dummy value "dontcare". It is to this complex predicate that the predicate formed by lambda-abstraction over "trvar" is passed. This "trvar" predicate is:

```
(lambda trvar (in Frederck IO))
```

Now, since there is no trace in the clause, the variable "trvar" does not appear anywhere in its translation and this lambda-abstraction over "trvar" is therefore vacuous. The effect of applying the OPTWH translation to it is to apply it to the value "dontcare". The successive simplification steps are shown:

```
((lambda (p) preds (p dontcare)) (lambda (trvar) (in Frederick IO)))

=>((lambda (trvar) (in Frederick IO)) dontcare)

=>(in Frederick IO)
```

Thus the final translation of the yes-no question "Is Frederick in the Indian Ocean" is the expression:

```
(in Frederck IO)
```

which is what we want.

Finally we have WH questions which do contain a trace, where the WH NP is not subject position. An example is "What ship did Jones deploy?" Here the non-empty OPTWH is generated by the rule:

```
OPTWH → WHPHRASE
semantics
(lambda (w) (lambda (p) preds (pred-to-set (w p))))
```

When this is applied to the translation of the WH NP "What ship" we get the following translation for the non-empty OPTWH:

```
(lambda (p) preds (set x ships (p x)))
```

This is a function that maps predicates to the set of ships for which the predicate is true.

The translation of the rest of the question clause—VP plus NP trace—is:

```
(lambda (trvar) (deploy-trans Jones trvar))
```

Note that this time the lambda-abstraction is not vacuous. This is a predicate returning true for an argument if Jones deployed it. The OPTWH translation is then applied to this argument to obtain the translation of the whole clause. We have, in various simplification steps:

```
((lambda (p) preds (set x ships (p x)))
    (lambda (trvar) (deploy-trans Jones trvar)))

=> (set x ships ((lambda (trvar) (deploy-trans Jones trvar))) x)

=> (set x ships (deploy-trans Jones x))
```

Thus, the final translation of "Which ships did Jones deploy?" is:

```
(set x ships (deploy-trans Jones x))
```

which is just what we want.

3.2.4.5 Complement Clauses and Traces

The complement clauses are divided into several sub-classes based on the value of the feature "comptype". The following are relative clauses, and are translated by predicates on individuals or groups of individuals:

THATCOMP "that is in the Indian Ocean" **WHCOMP**
"which is in the Indian Ocean" **NOCOMP**
"who loves Mary"

The following are translated by predicates of predicates (generalized quantifiers):

THANCOMP "than Frederick is" **ASCOMP**
"as fast Frederick is"

All these subclasses of complement clause are generated by the following rule:

```
S → COMP NP VP OPTSADJUNCT
semantics
(meta-lambda ($comp $np $vp $oa)
   (lambda (trvar) (union gts preds) ($oa (intension ((q $np) $vp)))))
```

All traces in our system are semantically translated into a distinguished variable, "trvar". The abstraction over "trvar" turns the normal propositional treatment of the clause into a predicate of whatever the trace is. In the case of relative clauses, this trace is a noun-phrase translation—hence these clauses are translated as regular predicates. In the case of THANCOMP and ASCOMP, this trace is an adjective-phrase translation—hence these clauses are translated as predicates of predicates.

3.2.5 Adjective Phrases

3.2.5.1 Simple ADJPs and Their Structure

Adjective phrases (ADJPs) have the semantics of one-place predicates in our system. The most basic kind of ADJP is generated by the following rule:

```
ADJP → ADJ-BAR
semantics
(lambda ($a) $a)
```

This rule builds an ADJP directly out of an ADJ-BAR, passing its semantic translation along as the translation of the entire ADJP. The ADJ-BAR is in turn built up out of an ADJ and whatever complement phrases the ADJ may specify (if any). In this way it is analogous in structure and semantics to VP. For example, the adjective "fast" does not take complements, and an ADJ-BAR is built directly from it by the rule:

```
ADJ-BAR → ADJ
semantics
(meta-lambda ($a) $a)
```

where the semantics of the non-complement-taking ADJ is stipulated to be a predicate, and is consequently passed along as the translation of the ADJ-BAR.

In contrast, adjective "close" sub-categorizes for a "to"-complement phrase: the ADJ-BAR "close to Vinson" is built by the rule:

```
ADJ-BAR → ADJ CASE-PHRASE
semantics
(meta-lambda ($a $cp) (lambda (x) ((q $cp) (lambda (y) ($a x y)))))
```

There are various other sub-categorizations for ADJ which are paired with various semantic types, again, in analogy with the VP.

3.2.5.2 Comparative and Superlative forms of ADJ

Comparative and superlative forms of an ADJ can be generated in two ways: directly through the morphology ("faster", "fastest") and by rule ("more truthful", "most truthful"). Whether done through the morphology or through the rule, the semantic effect is the same.

By convention, adjectives which can take a comparative or superlative form have a lexical semantic entry of the following form:

```
(lambda (x) gts (larger-than (<function> x) cvar))
```

where "<function>" is an underlying function of scale, "larger-than" a generalized predicate (discussed in section 3.3.2 and "c-var" a standard of comparison. For example, the lexical entry for "fast" would be:

```
(lambda (x) gts (larger-than (speed-of x) cvar))
```

The following is the rule for comparative:

```
ADJ → more ADJ
semantics
(meta-lambda ($m $a)
  (lambda (x) gts (larger-than ((uf $a) x) ((uf $a) cvar))))
```

The intensional operator of the language "UF" maps chosen predicates to functions on the same domain as the predicate. These predicates are just those that we have as the translations for adjectives that obey the convention above. The simplification transformation:

```
(UF (lambda ($v) gts (larger-than ($f x) cvar))) => $f
```

reduces the applications of "UF" to the underlying function from which the property is built.

When the comparative rule is applied to generate the ADJ "more fast" the resulting translation is:

```
(lambda (x) gts (larger-than (speed-of x) (speed-of cvar)))
```

The standard of comparision "cvar" now stands for whatever it is that "x" is "more fast" than. In the translation of "Frederick is more fast than Vinson", as we shall see later, the translation of "Vinson" is substituted for this variable. In the translation of "Frederick is more fast" it survives as a free discourse parameter.

The morphology of our system assigns separate semantics to the morpheme "-er" (and irregular forms thereof). This semantics is exactly the same as that given to "more" in the rule above, and all the above discussion applies to it in exactly the same way.

The following is the rule for the superlative:

```
ADJ → most ADJ
semantics
(meta-lambda ($m $a)
  ((lambda (P) preds
      (lambda (x) gts
         (element-of x
                    (first sup-num
                           (sort svar
                                 (lambda (y z) (svar svar)
                                       (larger-than ((uf P) y)
                                                    ((uf P) z))))))))
          $a))
```

The main body of this rule is a function from predicates to predicates; this is applied to the adjective translation ''a'' The new predicate returned yields true if its argument is a member of the set ''(first sup-num --)''.

This set is produced with the help of two operators and two free variables. The first operator, ''SORT'' takes a set ''S'' and a two-place antisymmetric predicate over the elements of that set ''r''. It produces a sequence of disjoint subsets of S that constitute an ordering of S under the relation r.[31] We need subsets of S as opposed to elements of S in order to deal with ''ties'' for a given ranking. For example, suppose Frederick to have a speed of 50 knots, Spica a speed of 40 knots, and Vinson and Kennedy each a speed of 30 knots. Then a ranking of these ships based on decreasing speed would be:

```
[{Frederick}, {Spica}, {Vinson, Kennedy}]
```

in which there is a tie for third place between Vinson and Kennedy, whose speed is the same.

The second operator ''FIRST'' takes an integer ''n'' and a sorted structure like this one. It delivers the first ''n'' root elements under this ranking, where the result is undefined if there are too many or too few for a given value of n. For example, for the sorted structure above if n is 1 the result is {Frederick}. If n is 2 the result is {Frederick,Spica}. If n = 3 the result is undefined (too many in third place) if n = 4 it is just the set {Frederick,Spica,Vinson,Kennedy} and for n = 5 and higher it is again undefined.

The free variable ''svar'' is the set over which superlativization is to be performed (in our example ships). The free variable ''supnum'' is just the integer variable ''n'' above and is the hook for ''the *three* fastest'', etc. Both ''supnum'' and ''svar'' are set by the rule for OPTNONPOSADJP (see section 3.2.2.5.

Thus, the translation of ''most fast''/''fastest'' is the following:

---

[31] More formally

$$s_1, \ldots, s_n$$

such that for each set s_i and for each e_j and e_k which are distinct elements of s_i neither $(r\ e_j\ e_k)$ or $(r\ e_k\ e_j)$ is true. Furthermore, for each set s_i and for each set s_j such that $j > i$ it is the case that for each element e_i of s_i and each element e_j of s_j, $(r\ e_i\ e_k)$

```
(lambda (x) gts
   (element-of x (first supnum
                        (sort supvar
                              (lambda (x y) (supvar supvar)
                                 (larger-than (speed-of x)
                                              (speed-of y)))))))
```

And the translation of "Frederick is fastest" is:

```
(element-of Frederick
            (first supnum
                   (sort supvar
                         (lambda (x y) (supvar supvar)
                            (larger-than (speed-of x)
                                         (speed-of y))))))
```

The variables **supnum** and **supvar** survive as free discourse parameters.

3.2.5.3 More Complex ADJPs

The rule producing an ADJP like "faster than Frederick" is:

```
ADJP → ADJ-BAR CASE-PHRASE
semantics
(meta-lambda ($ab $cnp)
   (lambda (x) gts ((q $cnp) (lambda (cvar) gts ($ab x)))))
```

Notice that lambda-abstraction over the cvar is performed in order to substitute the translation of the CASE-PHRASE for it. "Faster than Frederick" then has the translation:

```
(lambda (x) gts (larger-than (speed-of x) (speed-of Frederick)))
```

The rule producing ADJPs like "faster than Frederick is" is:

```
ADJP → ADJ-BAR S(thancomp)
semantics
(meta-lambda ($ab $s)
  (lambda (x) gts ($s (lambda (cvar) gts ($ab x)))))
```

The rule above performs an abstraction over cvar to form the predicate "ab-er than x", just as in the earlier example. Note that the constituent "than Frederick is" is an embedded THANCOMP sentence whose translation has the form (see the discussion on complement clauses in 3.2.4.5) of a predicate on predicates. The translation is:

```
(lambda (trvar) (trvar Frederick)
```

When this is applied to the predicate formed by abstraction over cvar, the effect is to apply that predicate to "Frederick". This gives the same result as before:

```
(lambda x (gts) (larger-than (speed-of x) (speed-of Frederick)))
```

Another type of ADJP has a unit noun phrase as a constituent: "five knots faster than Frederick". These are generated by the rule:

```
ADJP → (NP (unitnp)) (ADJ-BAR (comparative)) CASE-PHRASE
semantics
(meta-lambda ($np $adj-bar $case-phrase)
             (lambda (x) gts
                (forall y (set y1 things (inc y1 $case-phrase))
                   (forall x1 (set x2 things (inc x2 x))
                      (exists z $np (equal ((uf $adj-bar) x1)
                                           (plus z ((UF $adj-bar) y))))))))
```

The universal quantifications in this rule handle cases of plural arguments, as in "The frigates are five knots faster than the carriers". (It would require each of the frigates to be five knots faster than each of the carriers.) The existential quantification over "NP" enables the rule to handle such noun phrases as "more than five knots", which are translated as *sets* of amounts (those greater than five knots). The body of the existential quantification stipulates that the result of applying the underlying function of the ADJ-BAR to "x1" is equal to the result when it is applied to "y" plus the amount "z".

We can currently cover the following kinds of ADJPs and PPs

three miles long
(less than/more than) three miles long
(less than/more than) four knots faster
(less than/more than) four knots faster than Frederick
four knots faster than Frederick is
(less than/more than) four knots faster than Frederick is

three miles from Frederick
(more than/less than) three miles from Frederick
within three miles of Frederick
etc.

Cases of "how <adj>" are also handled, using the UF operator to derive the underlying function as the amount to be determined. This can deal with extraposition of complements to the adjective, as in "How close is Vinson to Frederick?".

3.2.5.4 Interaction with ADJ Subcategorization

The semantics for comparative can currently only handle "intransitive" adjectives—that is adjectives which do not take complements. This is because it relies on turning a one-place predicate into a two-place one, but adjectives that are not intransitive, such as "close" are already predicates of more than one argument place. Therefore, "closer than Vinson to Frederick" cannot currently be properly handled.

3.3 Lexical Semantics

This section discusses the lexical semantics component of the SLS system. The lexical semantic component is concerned with the specific meanings of a words in a subject domain, as a opposed to the manner in which these meanings are combined. We will present the general techniques which our work uses, and then show the application of it to various concrete issues, including relational nouns, noun-noun compounds, and collective/distributive quantification.

3.3.1 EFL to WML Translation

Up to now, we have treated example utterances as though each word could be assigned a unique descriptive constant as its semantic translation. This cannot always be the case, however, as words will often have multiple meanings. Consider the well-worn case of the word "bank": this can mean either the side of a river or a financial institution (and perhaps other things as well). Then too, different application domains can induce different senses on a word. For example, in the Resource Management Domain that much of our work has been devoted to, the collocational proper noun "Los Angeles" can refer to not only the city but a ship as well.

Certain very common words (or grammatical formatives) can have an even wider variation of meaning. The use of the preposition "on" in "the cat *on* the mat" clearly differs from its use in "the readiness *on* ASW". Other prepositions—such as "in", "with" and "of"—can also mean a great many different things in different contexts, as can the main verb "have" or the possessive "'s":

John's brother
John's car

Clearly, the semantics of "'s" is different between the first and second of these two examples: being in the first a kinship relation and in the second a relation of (most likely) legal possession.

The top-level description of this phenomemon is that different senses of a given word are evoked depending upon which other words it is combined with in an utterance. Our approach is to try to go to a slightly deeper level than this, and characterize which meanings of a word are evoked by which semantic *classes* of meanings of the other words. The chief reason for this (beyond its greater generality) is that it better allows the handling of *multiple* word ambiguity in an utterance, as seen in the phrase:

Los Angeles' speed

Here we have to contend simultaneously not only with the ambiguity of the possessive "'s", but also with the ambiguity (in our domain) of the term "Los Angeles".

The mechanism behind this is a two-level system of semantic interpretation: a structural level (described in the previous section) which always assigns a single semantic translation to any word, however ambiguous, and a lexical level, which expands this initial translation to the various terminal possibilities, filtering out those combinations which do not make sense. The structural level expresses its result in the logical language EFL (for English Formal Language) and the lexical level in the logical language WML (for World Model Language).

Translation is effected by a system of rules which associate with each constant of EFL one or more expressions of WML. A recursive-descent translation algorithm returns for each node of an EFL expression a set of WML translations. The translations for a constant expression are just those dictated by its WML translation rule. The translations for a complex expression are derived by combining, in a cartesian product fashion, the translations of the parts of the expression. At each level, the set of possible translations is filtered to remove *anomalous* translations—those which involve combinations of WML expressions with incompatible semantic types.

As an example, consider the declarative clause "Pacflt deployed Los Angeles". This would have the EFL interpretation:

```
(ASSERT (INTENSION (DEPLOY-EFL PACFLT LOS-ANGELES-EFL)))
```

Suppose the WML for our domain to include the predicate constant "DEPLOY", whose type is:

```
(FUN (TUPLES FLEETS SHIPS) TV)
```

and the following individual constants paired with types:

```
PACFLT            FLEETS
LOS-ANGELES-CA    CITIES
USS-LOS-ANGELES   SHIPS
```

where "FLEETS", "CITIES" and "SHIPS" are mutually disjoint atomic types. Finally, suppose the set of EFL to WML translations to include:

```
DEPLOY-EFL => DEPLOY

LOS-ANGLES-EFL => LOS-ANGELES-CA
               => USS-LOS-ANGELES
```

Then the combinatorially possible translations of the sub-expression "(DEPLOY-EFL LOS-ANGELES-EFL)" are:

```
(DEPLOY PACFLT LOS-ANGELES-CA)

(DEPLOY PACFLT USS-LOS-ANGELES)
```

Of these, the first is semantically anomalous, because CITIES and SHIPS are distinct, disjoint atomic types. The second is left, and survives the filtering to become part of the final result:

```
(ASSERT (INTENSION (DEPLOY PACFLT USS-LOS-ANGELES)))
```

3.3.2 Comparatives and Superlatives

An important application of the separate EFL level is seen in words whose most natural semantic translations have *internal* points of ambiguity. Consider the following:

The Sears Tower is *higher* than the World Trade Center
Frederick's fuel capacity is *higher* than Vinson's
Vinson's readiness is *higher* than Frederick's

Clearly the word "higher" must mean something quite different in all three sentences. Moreover, the same kind of variation in sense is seen in other inflectional forms of the adjective:

Frederick's fuel capacity is *high*
The Sears Tower is the *highest* building

We would not want to have to handle this by having three semantic entries apiece for "high","higher" and "highest".

We have already shown how the semantics of the various inflected forms of an adjective can be derived from semantics of its positive form by giving it a translation that fits the schema:

```
(lambda (x) ($r ($f x) $c))
```

where "$f" is an underlying function, "$r" a relation and "$c" a standard of comparison. Which sense of the adjective is meant then depends on the choice of this underlying function.

The most obvious choice of underlying function for "high" would be something like "HEIGHT-OF". This will apply just fine to the Sears Tower. But it will fall flat on its face when applied to readiness values and fuel volume amounts. One might widen the domain and range of the function to accomodate these new inputs, but this seems a strange thing to do: it would be changing facts about the *world* to accomodate facts about *language use*.

What seems better is to define a linguistic notion of "height" and a linguistic notion of "greater than" which can be expanded to the various real world notions as needed. The semantic entry for "high" is thus:

```
(lambda (x) gts (LARGER-THAN (HEIGHT-EFL x) cvar))
```

Where "LARGER-THAN" and "HEIGHT-EFL" are both EFL constants with the translations:

```
HEIGHT-EFL => HEIGHT-OF
           => (lambda (x) VALUES x)

LARGER-THAN => (lambda (x y) (numeric numeric) (greater x y))
            => HIGHER-READINESS
```

where VALUES is defined as the union type of NUMBERS, AMOUNTS and READINESS-VALUES and NUMERIC is the union type of NUMBERS and AMOUNTS alone. "HIGHER-READINESS" is a WML predicate applying to pairs of readiness values, returning TRUE for the pair <C1,C2>, FALSE for the pair <C2,C1> etc.

"The Sears Tower is higher than the World Trade Center" thus has the following EFL translation:

```
(LARGER-THAN (EFL-HEIGHT SEARS-TOWER) (EFL-HEIGHT WORLD-TRADE))
```

The first translation of "HEIGHT-EFL", "HEIGHT-OF" fits both "SEARS-TOWER" and "WORLD-TRADE" while the second translation does not. We have then the partial translation:

```
(LARGER-THAN (HEIGHT-OF SEARS-TOWER) (HEIGHT-OF WORLD-TRADE))
```

The function "HEIGHT-OF" returns a length amount, which by definition is a member of NUMERIC. Therefore the first translation of LARGER-THAN applies but the second does not, leaving us with the final result:

```
(GREATER (HEIGHT-OF SEARS-TOWER) (HEIGHT-OF WORLD-TRADE))
```

The sentence "The Sears Tower is the highest building" has the EFL translation:

```
(equal SEARS-TOWER
       (first 1 (sort BUILDINGS
                      (lambda (x y) (BUILDINGS BUILDINGS)
                              (LARGER-THAN (HEIGHT-EFL x)
                                           (HEIGHT-EFL y))))))
```

By exactly similar translation steps HEIGHT-EFL and LARGER-THAN are expanded to the same expressions as before.

The sentence "Frederick's readiness is higher than Vinson's" can be shown to have the final WML translation:

```
(HIGHER-READINESS (READINESS-OF FREDERICK) (READINESS-OF VINSON))
```

This shows how the EFL level captures the same kind of sense variation across the spectrum of positive, comparative and superlative forms of an adjective.

3.3.3 Relational Nouns

In this subsection we show the application of the EFL level of processing to another problem: that of "relational nouns".

Earlier we contrasted two uses of the possessive:

```
John's brother
John's car
```

The first case seems different from the second; the relation to which the possessive is to be translated seems expressed by the noun "brother" itself. Nouns like "brother" (or "speed" etc.) are what are commonly called "relational" nouns: their semantics has the character of a relation mapping some argument (here, "John") to some other individual (here, the person who is John's brother). The semantics of relational nouns (reported in [13]) was a major topic of research in our work, and we will discuss them in some detail.

3.3.3.1 The Subcategorization Approach

One possible approach to relational nouns is to treat them like verbs, which sub-categorize for their arguments as regular complements. As was noted in earlier discussion on noun phrase semantics (3.2.2), their are some nouns for which this seems the correct approach:

The *list* of the spies

The noun "list" is here construed as taking an argument CASE-PHRASE; it is translated as a function taking the semantics of this CASE-PHRASE as an argument to make up the meaning of the whole NP. Note that while "list" does not *require* this argument:

He's making a list and checking it twice

this optionality can be accomodated by having a second subcategorization that does take a complement. This subcategorization would be the one present in:

The spies's list

which has a completely different meaning: that of a list (presumably of vital secrets) in the spies possession.

This pattern of distribution certainly does not hold of all relational nouns. Consider:

The picture of John
The picture that John has
John's picture

The first refers to a picture representing John, while the second can only refer to a picture that John possesses. The third, on the other hand, is ambiguous: it can have either meaning. It is not standard to consider "John's" to be a complement of the noun "picture" here, yet semantically it is apparently able to function as one: it fills in the argument place to "picture".

Other relational nouns are still freer in their distribution. Consider the syntactically parallel examples:

The speed of Frederick
The speed that Frederick has
Frederick's speed

All three of these have exactly the same meaning, even though "that Frederick has" cannot possibly be a complement. What is more, there are other constructions, such as:

The ship that has a speed of 30 knots
The ship with a speed of 30 knots

in which the complement-taking noun is inside an NP of its own, yet still is able to receive the argument "The ship". We therefore concluded (in [13]) that the notion of "argument" here is semantic and not syntactic in nature.

3.3.3.2 The SLS Treatment

Our treatment was to allow these semantic arguments to relational nouns to be provided by "connective" elements such as "have", "'s","of" and "with". These are rendered as EFL constants which must be given the appropriate WML translation to complete the meaning of the utterance. For example, the noun phrase "John's mother" might be rendered in EFL as:

```
(the (set x MOTHERS (OF x John)))
```

The first question is what the constant "MOTHERS" above denotes: that is, what semantic translation to give relational nouns. One possible answer is to make them the "range" of the underlying relation; that is to "mother" assign the set of women who have at least one child. Then the EFL representation of the NP "The mother of John" would be:

```
(the (set x (range mother-of) (of x John)))
```

and the obvious translation of "OF" would be:

```
OF =>  (lambda (x y) ((range mother-of) people)
                     (equal x (mother-of y)))
```

which can be felicitously applied just in case the first argument is a member of the set of mothers[32].

This solution will come to grief, however, if our system also encompasses the relational noun "wife", with a similar translation rule:

```
OF => (lambda (x y) ((range wife-of) people)
                    (equal x (wife-of y)))
```

Supposing some woman, say Jane, is John's wife and that she has children. Then she is a member of both the set "(range mother-of)" and the set "(range wife-of)" and can thus pass through both translation rules. The statement "Jane is John's wife" will then have two translations:

```
(equal Jane (wife-of John))

(equal Jane (mother-of John))
```

the first of which evaluates to TRUE and the second to FALSE. This is surely not correct.

Our solution to this problem was to make the translations of relational nouns include not just the range of the relation, but the entire relation itself. The translation of "wife" would be a set of ordered pairs such that the first element is the wife of the second. In our example, it would include the ordered pair:

```
<Jane, John>
```

Finding the right translation for "OF" ("HAVE", "WITH" etc.) now means connecting the second argument to "OF" with the second elements of these ordered pairs. The rule for "OF" now becomes:

```
OF => (lambda (x y) ((tuples females people) people)
                    (equal (elt 2 x)  y))
```

This rule suffices to handle both the "mother of " and "wife of" cases; the information distinguishing them is built into the first argument.

To handle "the speed of Frederick", "Frederick's speed" and "The speed that Frederick has" one could have the following rules:

```
OF => (lambda (x y) ((tuples (amounts speed-measure) ships) ships)
                    (equal (elt 2 x) y))

HAVE => (lambda (x y) (ships (tuples (amounts speed-measure) ships))
                      (equal (elt 2 y) x))
```

Relational nouns with more than one argument, such as "distance" in "the distance from Frederick to Vinson" could also be accomodated:

```
FROM => (lambda (x y) ((tuples (amounts length-measure)
                               (tuples locatables locatables))
                       locatables)
                      (equal (elt 1 (elt 2 x)) y))
```

[32]Recall that our logic allows not only types, but arbitrary set-denoting expressions to appear in the range restriction of a quantifier. Even if the type system is not fine-grained enough to distinguish between mothers and other females an incorrect argument could still be detected at evaluation time, and rejected on that basis.

In later work, we modified this treatment. First, we substituted a novel type operator, SUB-ENTITIES, for TUPLES in the rules above in order to distinguish these constructions from uses of TUPLES for other purposes. The type restriction on the first argument above was now rendered:

```
(SUB-ENTITIES (amounts length-measure) (tuples locatables locatables))
```

where the first argument type "(amounts length-measure)" is referred to as the ENTITY type and the second argument type "(tuples locatables locatables)" is referred to as the INDEXER type.

Secondly, we modified the treatment to allow for a more general and schematic kind of relational translation rule. Instead of having two separate translation rules for "OF", one for "mother"/"wife" and the other for "speed", we had one:

```
OF => (lambda (x y) ((sub-entities anytype anytype) anytype)
                    (ANY-ARG x y))
```

"ANY-ARG" is an EFL constant which is expanded not by translation rules, but by a special procedure for it which was added to the EFL to WML translation algorithm. This special procedure searches the INDEXER type of the first argument to "ANY-ARG" for "slots" in which the second argument to "ANY-ARG" could fit. Appropriate expressions, similar to those in the body of the rules above, are then constructed and returned as the expansion of the "ANY-ARG" application.

A number of other EFL constants have translation rules containing the ANY-ARG hook. Among them are "HAVE", "ON", "WITH" and "PREMOD", the place holder for nominal compounds.

The "PREMOD" relational translation enables the system to cope with such (rather telegraphic) phrases as "Frederick speed". More commonly, this feature is invoked for such utterances as "equipment readiness", where the word "equipment" may be replaced by "training", "personnel", "asw" and other terms denoting an "area" on which readiness is computed.

3.3.3.3 Problems with the SLS treatment

In experiments in coupling the SLS semantics to the output of the speech system we noticed that the following rather odd noun phrase was being accepted and interpreted:

Frederick's Kirk readiness

"Frederick" and "Kirk" are both the names of ships. The interpretation that was being given was:

```
(the (set x (reify readiness-of)
            (and (equal (elt 1 (indexer x)) Frederick)
                 (equal (elt 1 (indexer x)) Kirk))))
```

This expression denotes the member of the extension of the function "readiness-of" such that the first element of its right-hand side (its "indexer") is the individual denoted by "Frederick" and such that it is also the individual denoted by "Kirk". (The operator "reify" takes a function and produces the set of sub-entities corresponding to the elements of its extension.) This is clearly an unsatisfiable condition since "Frederick" and "Kirk" denote entirely different individuals.

A similar situation is seen in the interpretation of the noun phrase "Frederick's distance from Vinson". The function translating "distance" has two arguments (a "from" and a "to" as it were) that the term "Frederick" fits into. Thus the expansion of the "ANY-ARG" for the possessive "'s" gives rise to two possiblities, and thus two possiblities for the translation for the entire NP:

```
(the (set x (reify distance-between)
            (and (equal (elt 1 (indexer x)) Vinson)
                 (equal (elt 2 (indexer x)) Frederick))))

(the (set x (reify distance-between)
            (and (equal (elt 1 (indexer x)) Vinson)
                 (equal (elt 1 (indexer x)) Frederick))))
```

Of these, only the first expression is the correct translation. The second translation has the same problem as seen above: it asks for an element of the function extension in which the first argument is both Vinson and Frederick.

We were able to get around the problem for these particular examples by having a simplification transformation:

```
(and (equal $term $a) (equal $term $b)) => FALSE
```

where `$a, $b` are logical proper names[33] and `$a` is distinct from `$b`

which could recognize the unsatisfiablity of these conditions, enabling the system to subsequently apply the transformations:

```
(set $v $s FALSE) => NULL-SET

(the NULL-SET) => DENOTATIONLESS
```

and reject the inappropriate interpretations.

A similar problem still exists, however, for the query "what if Frederick had Vinson's capabilities". This is translated by the system into an expression that can rougly paraphrased as "what if every element of the capabilities relation that has Vinson as a first element also had Frederick as a first argument?". Similar simplification mechanisms would reduce this expression to:

```
(what-if FALSE)
```

which is the system's way of saying that this is an "impossible counterfactual".

3.3.3.4 Conclusion

The semantic incoherence of such NPs as "Frederick's speed of Vinson" and "Frederick's Vinson speed" indicates that there is an "exclusion principle" at work with relational nouns: the same semantic argument slot can only be filled once. In ordinary mathematical notation this is an issue which never even arises, since the very syntax of function application does not provide an opportunity for filling in an argument more than once. Our current treatment of relational nouns is different, since it "fills in an argument" by predication on the extension of the

[33]Logical proper names are designated individual constants such that distinct logical proper names necessarily denote distinct individuals

function and such predications can always be added to one another arbitrarily. Ultimately, the result is the same: applying the function more than once leads to a non-meaningful result. At issue is at what level of processing this condition is detected.

The "exclusion principle" is evidence for a more syntactic treatment of relational nouns, something more like the treatment of verb subcategorization, in which it has long been recognized that two different phrases cannot enter into the same syntactic relation with the verb. The earlier argument against such a syntactic treatment was that the verb "have" could supply arguments to the noun. But is not clear that the same theory must explain argument-passing by "have" and by other, NP-internal elements such as "'s" and "of".

Consider the earlier example, "What if Frederick *had* Vinson's capabilities". The fact that "Vinson" has filled in the argument place for "capabilities" is not an obstacle to our understanding of the sentence. The "exclusion principle" seems to only operate inside NPs and not outside of them. Therefore there is no reason to assume that "'s" and "of" must be treated in the same way as main verb "have".

3.3.4 The Translation of Nominal Compounds

We have already seen how a nominal compound like "equipment readiness" or "Frederick speed" is given a relation-argument interpretation. Another schematic expansion for PREMOD is "SUB-CATEGORY". It handles compounds like "training resource", where "TRAINING" is a subcategory of "RESOURCES" and "asw mission", where "ASW" is an individual member of "MISSIONS". Working together, the two expansion schemas for PREMOD can succesfully deal with long-winded compounds such as:

"equipment resource area readiness rating"

which are fairly common in our corpus.

The SUB-CATEGORY function can also combine a relational construction and its value, as in "c3 readiness", "30 knot speed" etc. The interpretation of the latter is:

```
(set x (reify speed-of) (equal (entity x) (amount 30 knots)))
```

One problem was noted that arose from the left-branching character of the nominal compound rule. A compound like "c3 equipment resource readiness" is given a structure:

[[[c3 equipment] resource] readiness]

There is no way to interpret the compound [c3 equipment] and so the entire compound cannot be interpreted by the system. This can be compared to the bracketings

[[c3 readiness] rating]

and

[[equipment resource] readiness]

which can be interpreted.

Other, non-schematic translations for PREMOD were provided that enabled the system to handle such compounds as:

diesel ships
Atlantic Ocean frigates
Midpac cruisers
link-11 ships

among others.

No more general mechanisms for compound-handling were attempted, nor was any method of handling "frozen" compounds (except for the mechanism of collocation) incorporated.

3.3.5 Plurals and Distributive/Collective Quantification

Quantification is *collective* when a predicate is applied to an entire set at once, instead of to each of its elements separately. Consider the following sentences paired with their semantic translations:

The boys walk
```
(forall x (the-plural BOYS) (WALK x))
```

The boys gather
```
(GATHER (the-plural BOYS))
```

The second sentence has a very different-looking translation from the first because of the semantic nature of the verb "gather": an individual cannot "gather" by himself. The first sentence exhibits distributive quantification, the second, collective quantification.

Verbs are collective or not with respect to their individual argument places, as the following example derived from our corpus makes clear:

Frederick's capabilities *include* link-11 and harpoon

```
(and (element-of LINK-11 (CAPABILITIES-OF FREDERICK))
     (element-of HARPOON (CAPABILITIES-OF FREDERICK)))
```

where the collectively interpreted argument is the subject "Frederick's capabilities". In addition, the argument given to a collective argument place need not be syntactically plural as long as it is a semantic collective:

The battle group *includes* Frederick

```
(element-of Frederick (SHIPS-OF (the BATTLE-GROUPS)))
```

The structural semantics of the SLS system translates all plural definite NPs (including possessive NPs like "Frederick's capabilities") as sets. Therefore, at the EFL level, the translations of "The boys walk" and "The boys gather" look alike: the EFL predicate translating the verb is applied directly to the set:

```
(WALK-EFL (the-plural BOYS))
```

The EFL to WML translation stage then makes the decision between distributive and collective quantification on a per-predicate basis. The information for doing so is encoded in the EFL to WML translation rule itself:

```
WALK-EFL => (lambda (x) (forall y (PARTS (setof x)) (WALK y)))
```

The "PARTS" function is another EFL constant that is responsible for carrying out distribution. It is recursive in nature, and has the expansions:

1. (PARTS x) => x if x an individual
2. (PARTS s) => (collect s PARTS) if s a set
3. (PARTS s) => (U (collect s PARTS)) if s a set
4. (PARTS x) => (F x) where F a "membership function"

The fourth expansion handles examples like "The battle group *includes* Frederick", where the collective entity "the battle group" is expanded to its extensional membership.

The expansion of PARTS was made recursive in order to handle the problem of "multi-level plurals", such as "the boys and the girls", "the juries and the committees" etc. As discussed in [43] collective verbs applied to such NPs have an option of "partial distributivity" over the outer levels of plural structure. An example would be "The boys and the girls gather"; this has two possible interpretations:

```
(GATHER (union (the-plural BOYS) (the-plural GIRLS)))

(and (GATHER (the-plural BOYS))
     (GATHER (the-plural GIRLS)))
```

The PARTS function can handle an arbitary number of levels of plurality, producing a correspondingly arbitrary number of WML translations for an EFL predicate.

3.3.5.1 Handling the sub-entities

To accomodate sub-entities, we introduced another expansion to the PARTS function:

```
PARTS => (ENTITY x)  if x a sub-entity
```

The type operator "GROUPS" was also introduced. If α is a type, "(GROUPS α)" forms the infinite union type whose members are the type "S" inductively defined:

α is a member of **S**

if β is a member of **S**,
then **(sets** β**)** is a member of **s**
and **(sub-entities** β **anytype)** is a member of **s**

if $\beta_1,\ldots,\beta_n$ are members of **S**
then **(tuples** $\beta_1, \ldots, \beta_n$**)** is a member of **s**

3.3.6 EFL and Incremental Semantics

This work did not attempt an incremental version of semantic interpretation where translation would be made all the way to WML as each sub-constituent was parsed

The original version of EFL to WML translation [7] does not particularly conflict with such an incremental approach. A difficulty arises, however, when one includes the modifications to these ideas that we have made in our work. If a given verb is made non-finitely ambiguous to handle infinitely various possiblities of plural arguments, logically it should have an infinite number of WML translations when considered in isolation. Similarly, if the preposition "of" can connect its object to any suitable argument place in a sub-entity structure that can, in principle, have an arbitrary number of argument places it, too, must have have an infinite number of WML translations when considered in isolation.

Both devices—"PARTS" and "ANY-ARG" depend on having a completed structure to look at. This is because they are schemas—more like branching categories than actual functional constants. They stand in for non-finite sets of expressions. Using them in an incremental approach to semantic interpretation would require finding a way to accomodate such non-finite sets in the incremental algorithm in such a way that the algorithm would attempt to resolve this non-finite ambiguity only when all information was in place, and not before. Of course for utterances which are not completed sentences—such as the VP "include link-11"—this ambiguity could not be resolved and would have to remain in the translation.

An alternative approach would be a separated architecture in which incremental semantics was done in one pass to resolve traditional kinds of ambiguities, such as that seen in the two senses of "Los Angeles" in the example above. This incremental step would be allowed to produce multiple interpretations for the same phrase and and thus would not need to produce an EFL representation (or at least not an EFL representation of quite the same kind). A second pass would then deal with quantificational matters, here construed (perhaps reasonably) to include the collective/distributive distinction. If one wanted to, he could still regard the logical expressions input to this second pass as being ambiguous, and therefore as a kind of EFL.

There remains the question of internal ambiguity raised in the sub-section on comparatives and superlatives (3.3.2). Recall there that the word "high" was regarded as having "internal" points of ambiguity in its lexical meaning: what scale-function to use and what comparision relation to use. Different choices were appropriate to different inputs: fuel-capacities, readiness-values buildings. Note this is a particular kind of ambiguity, however. No possible input could fall into more than one of these categories. Accordingly, we might use a different device than ambiguity—a "function-union"

```
HEIGHT := (F-UNION HEIGHT-OF (lambda (x) VALUES x))

LARGER-THAN := (F-UNION HIGHER-READINESS
                        (lambda (x y) (NUMERIC NUMERIC)
                                      (greater x y)))
```

The operator F-UNION takes an arbitrary number of functions and produces a new function defined to have the

extension that is the set-union of the extensions of the argument functions. Obviously, this must still be a function, i.e. the argument functions must not disagree in any of their assignments. This condition is guaranteed in the case that the functions have mutually disjoint domains, as is the case above.

When applied to arguments of a particular type, such F-UNION expressions can be simplified be eliminating those of its member functions whose domains are disjoint with the type of the argument. By these means the translations in 3.3.2 can be produced without recourse to EFL-type ambiguity.

This technique is interesting from a general knowledge-representation point of view because it allows us to construct the intuitively needed generalized version of say, "GREATER" without positing in our domain model a primitive element LARGER-THAN that subsumes GREATER and HIGHER-READINESS. The relation LARGER-THAN is not primitive because it has been defined by F-UNION. It has no mysterious properties.

In conclusion we might also note that the F-UNION technique is also applicable to our collective/distributive question, since the different domains there—individuals, sets, sets of sets etc.—are all mutually disjoint.

3.4 Simplification

Simplification can be invoked for expressions at every level of processing in the system. It utilizes a set of language-level independent global transformation rules which can be added to by system builders.

3.4.1 The Rules

3.4.1.1 Basic Patterns

These rules consist basically of initial and final expression patterns. Meta-variables in these expressions, prefixed by a "$", range over expressions of the language. Instantiation of a rule is performed by matching the input pattern to the expression to be simplified and producing the appropriate subsitution instance of the output pattern. Example:

```
(set $v $s true) => $s
```

This transformation turns a set-abstraction with a vacuosly true body into the range of its variable. If applied to the expression below:

```
(set x SHIPS true)
```

the instantiation of its variables would be:

```
(($v . x) ($s . SHIPS))
```

and the final output would be:

```
SHIPS
```

Provision is made for meta-variables that match more than one element. Example:

```
(and TRUE . $rest) => (and . $rest)
```

The variable "$rest" has the instantiation "(($rest p q r))" when this transformation is applied to the expression:

```
(and TRUE p q r)
```

producing the output:

```
(and p q r)
```

For commutative operators like AND, matching is order-independent. That is, the above transformation could have been just as easily applied to "(and p q TRUE r)", with identical results (modulo irrelevant ordering differences in the final expression).

3.4.1.2 Optional Constituents

Transformations have two optional elements. The first is a test, written in LISP code, which is applied as a further condition to the firing of the transformation if its input pattern matches. Example:

```
((element-of $term $set)
 TRUE
(and (type-exp? $set)
     (sub-type? (typeof $term) $set)))
```

The LISP test is the third element of the list; it stipulates that "$set" be a type expression and that the type of "$term" be a sub-type of this term. If that is the case, the semantics of the type system guarantees that "$term" is an element of "$set" and the entire expression can be reduced to TRUE. Meta-variables are implicitly quoted in such LISP code.

The other optional element is a meta-variable which does not occur in the input pattern but which does occur in the output pattern paired with LISP code. Its effect is to unify this variable with the result of the LISP expression, which then is used in the instantiation of the output pattern. Probably the most important example is in the transformation implementing the beta-reduction simplification of lambda-calculus:

```
((apply (LAMBDA ($v) $s $body) $arg) ;;BETA-REDUCTION
 $result
 (and (compatible? (typeof $arg) (typeof $v))
      (or (not (in-intensional-scope? $v $body))
          (modally-closed? $arg)))
 ($result (if (type-exp? $s)
              (subst-var $arg $v $body)
              (make IF-THEN
                    predicate (make ELEMENT-OF
                                    item $arg
                                    set $s)
                    expl (subst-var $arg $v $body)))))
```

This transformation substitutes "$arg" for "$v" in "$body" just in case certain conditions are fufilled—namely that types be compatible and that the variable not be within the scope of an intension[34].

[34]Currently this requirement has been suspended—see discussion of intensionality in 3.2.3.6

3.4.2 The Algorithm

The algorithm is implemented by the LISP function SIMPLIFY. It is a recursive descent algorithm, meaning that the sub-expressions of an expression are simplified before the expression itself. Additionally, when a simplification transformation fires it re-invokes the simplifier on the instantiation of the output pattern.[35] Assuming there are no cycles in the set of transformations, the process ends when no more transformations can be instantiated. An example of a very simple cycle might be a pair of transformations:

```
(exists $v $s $formula) => (non-empty (set $v $s $formula))

(non-empty (set $v $s $formula)) => (exists $v $s $formula)
```

As a technical matter of principle, detections of such cycles might become difficult, since one can have more than just two transformations involved, etc. In practice, using a set of ~100 transformations constructed with modest care, we have not encountered any problems with such cycles.

Note that the commutative property of the matcher coupled with the double-recursion of the algorithm allows the system to perform certain tasks over and over until they are finished, like "get the TRUEs out of the conjunction". The result of simplifying:

```
(AND p TRUE q TRUE r TRUE s)
```

is finally just:

```
(AND p q r s)
```

Such commutative matching requires that multiple matches be returned for consideration at any level of the match. An example:

```
((apply (f-union $f . $other-fs) $term)
 (apply (f-union . $other-fs) $term)
 (disjoint-types? (domain-type (typeof $f))
                  (typeof $term)))
```

This transformation can be used to succesively get rid of functions in a union of functions when their types do not allow them to be applied to a given argument. Notice, however, that if just a single match for "$f" and "$other-fs" is returned the transformation will only succeed if "$f" happens to have the incompability in question. If multiple matches are returned, all possibilities are considered when it comes time to apply the LISP test.

[35]This double recursion, due to a suggestion by Marc Vilain, turns out to be considerably more efficient than embedding the recursive-descent algorithm in a continous iteration, our original method.

3.4.3 Interesting Applications for Simplification

Simplification has more uses than just taking care of cleaning up logical forms that were egregiously complicated to begin with. Because it applies tautologies, which in an intensional/modal logic are a sub-species of necessary truth, it can be used to extract more information from a query than one would otherwise be able to.

For example, the (granted rather silly) query "Are all submarines ships?" can be reduced by simplification and the type system to just:

```
(query true)
```

No evaluation is required to answer this question with this logical form; the answer is always "yes" no matter what the current state of the world is[36] Recognition of this fact enables us to answer more than just "yes": it enables us to say "necessarily, yes", a much stronger and possibly more useful statement.

Of course questions with necessary answers are unlikely to be asked most of the time, especially by people who have some knowledge of the application domain. When a question is reducible to a necessary answer this may then be an argument against the interpretation of the question that led to this result.

Consider the question "Is Frederick C3?". The word "C3" appears as both an adjective and a noun in our lexicon, as it must in order to handle both this question and the question "What ships have a readiness of C3?". Therefore, "Is Frederick C3?" will receive two parses: one where "C3" is an adjective (the correct parse) and one where "C3" is a noun (the incorrect parse). The latter will be represented by the logical form:

```
(equal Frederick C3)
```

which can be paraphrased as asking "Is Frederick identical to the readiness value C3?". This is an absurd-seeming question, but not a nonsensical one in our framework: it has an answer, yes or no. This interpretation cannot be ruled out if only selectional restrictions are considered[37].

This expression can, however, be reduced by the transformation:

```
(equal $term1 $term2)  => FALSE
       when (disjoint-non-empty-types? (typeof $term1) (typeof $term2)
```

We then have the entire expression representing the query reduced to

```
(query FALSE)
```

and this can be devalued with respect to the interpretation arising from other parse of the query:

```
(query (equal (readiness-of Frederick) C3))
```

which does not have this problem.

---

[36]Remember that by the semantics of our type system, the world can only be in states that are allowed by the type system.

[37]Of course one might stipulate that the meaning of "be" in a predicate nominal ought to require that the types of its arguments be "similar" in one way or another, perhaps by being non-disjoint. Consider however, an interaction where the user, referring to a cluster of dots on a screen that represent a group of islands and not knowing what they are, asks "Are those ships?"

A more general version of this heuristic is that is not desirable to have descriptive constants (as opposed to formal constants like FALSE and TRUE) disappear during logical simplification. This comes down to a kind of Gricean maxim of quality: if they were not necessary in formulating the meaning of the query, why would the speaker have included them?

Obviously, the two benefits cited in this section—giving useful information about the necessity of an answer and eliminating absurd but not meaningless interpretations—collide to some extent. We would not want to the system to forbid a "generic" question on the grounds that no one would be dumb enough to ask it. On the other hand, we would not want it to assume that the user might be so out of touch with his world as to imagine that the set of ships and abstract values could co-incide. What is needed, it seems, is a notion of strength or distance of disjointness: not in the underlying mathematics, but in the way that individual facts of disjointness are expressed. This could be represented by lesser or greater distances between concepts in a hierarchy, etc.

3.5 Answering Component

Our overall system design includes a separate language step called DBL, used to access the tables of a database. A DBL step is in general necessary because the structure of the data files in the target system may be very different from the logical structure of the real world domain.

In our work we chose to build a "toy" database to pose queries to. To simplify matters, we essentially dispense with the DBL level, requiring the data file structure of our toy database to correspond to the logical structure of the WML domain model. This database is represented in the language CVL.

CVL (for Canonical Value Language) is a very simple logical language whose constants are all required to be what are called "logical proper names". Logical proper names are constants with two special properties. First, distinct logical proper names necessarily denote distinct individuals in the domain. Second, a logical proper name always denotes the same individual regardless of the time and world index of evaluation.

Examples of logical proper names are the numerical constants: "1", "2" etc. These have the two properties required (and it is a good thing, otherwise we couldn't perform reliable arithmetic computations using these symbols!) They are also formal constants, which means that they are built into the language and are domain-independent. In our work we have chosen to include domain-dependent, descriptive logical proper names as well. Examples are "FREDERICK" (standing for a particular ship) "INDIAN-OCEAN" (standing for a particular region) and "C3" (standing for a particular readiness value).

The principal operators of CVL are "SETOF", which takes an arbitrary number of arguments and constructs an explicit set containing them, and "TUPLE" which takes an arbitrary number, n, of arguments (in some order) and constructs an n-tuple of them. Data tables are then represented as

```
(SETOF (TUPLE FREDERICK C3)
       (TUPLE VINSON c4)
         ...
       )
```

i.e. as a set of tuples.

The task of evaluation is thus to take a WML expression and deliver the CVL expression that corresponds to the denotation the WML expression. The method of evaluation is recursive: with each operator of WML is associated an evaluation rule (see the semantic framework section, 3.1.3) which computes the CVL value of the expression from the CVL values of its parts. The CVL value of terminal constants is assigned before hand as part of the database. For example, the value assigned to the constant "SPEED-OF" might be:

```
SPEED-OF =>
    (function
       (setof (tuple Frederick (amount 30 knots))
              (tuple Vinson (amount 40 knots))
                 ...))
```

("AMOUNT" and "FUNCTION" are both CVL operators, the latter taking a set of tuples and delivering the corresponding function whose extension they are.)

For more on CVL, see the document "A Manual for the Logical Language of the BBN Spoken Language System" [47].

4. Speech and Natural Language Integration

In the preceding chapters, we have described the natural language components of the BBN Spoken Language system: the parsing algorithm that uses the BBN ACFG (Chapter 2) and the semantic interpreter that derives a meaningful interpretation of text input (Chapter 3). In this chapter, we will describe our approach to integrating syntax and semantics with acoustic scoring for speech understanding.

The goal of speech understanding is to determine what was spoken and the corresponding meaning of the input utterance. To achieve optimal performance, i.e., the maximum correct understanding rate, we need to find the most likely word sequence consistent with syntax and semantics. This poses the problem of a large search space which must be explored judiciously so that an utterance can be processed in a reasonable amount of time with reasonable computational resources.

There are several possible approaches to solving the speech understanding problem.

One possible approach, which we have demonstrated previously, is the serial connection. In this approach, speech recognition and natural language processing are performed serially and independently, with the speech recognition component computing the best scoring answer using acoustic models and its own language model, and then passing the answer to the natural language component for processing and interpretation. The critical problem with this approach is the possibility of a mismatch between the speech language model and the natural language grammar: the sequence of words recognized by the speech component could possibly fall outside the coverage of the natural language grammar, causing the system to break down altogether. Also, if the speech recognition component makes an error, there is little chance for recovery. Therefore, to have any chance of success, one needs to fully integrate speech and natural language, where integration means using same the language model to jointly perform speech recognition and natural language understanding in a single search space.

One approach to integration is to compile the natural language syntax and semantics into a single network such as a Finite State Automaton (FSA) or a Recursive Transition Network (RTN) appropriate for performing a top-down time-synchronous search to find the best hypothesis [12]. However, this assumes that such a network can in fact be built from the declarative unification grammar formalism of our natural language syntactic component and our semantics component. A close examination of our grammar reveals that the number of equivalent context-free rules needed to make an FSA network from our unification grammar (semantics not included) would run into the hundreds of thousands, and the number of arcs in this FSA network would be many times that size. No computer on the market or on paper today would have a virtual address capacity anywhere near this size, not to mention the paging penalties that would be incurred even if such a computer were available.

The approach that we have taken, then, is to perform parsing (in the natural language processing sense) on the speech input. This approach consists of a two step process. First, the speech component computes a very dense word lattice; all words that are plausible acoustically anywhere in the input utterance would be computed, with a

separate score for every starting and ending time. Given this word lattice, the natural language component can search for the most likely meaningful sentence as a path through the lattice.

As such, the problem can now be posed as a parsing problem solvable by parsing algorithms similar to the text parsing algorithm described in Chapter 2. Whereas the text parser takes one sentence as input, the "speech parser" takes the lattice of alternate word hypotheses, and finds in the lattice *all* grammatical sentences (using syntactic information only) and assigns each sentence an acoustic likelihood score. Henceforth, we shall call this speech parser the Lattice Parser.

Our extensions to the text parser to handle the lattice are as follows:

1. Input is a lattice consisting of *micro acoustic scores* $S(t_1,t_2 \mid w)$, each with a distinct *starting time* t_1 and *ending time* t_2. Neighboring acoustic scores belonging to a word w are grouped into an *acoustic segment*, where an acoustic segment now consists of a set of these *micro* acoustic scores $\{S(t_1,t_2 \mid w)\}$, called a *word theory* or *terminal theory*. Because an word theory is made up of acoustic scores that are adjoining in time, the same word w in V can have multiple word theories in the lattice corresponding to different occurrences of that word.
2. Because terminals have scores, dotted rules have scores associated with them:

   $(A \rightarrow B.CD, \{S(t_1,t_2)\})$

   where

   $(A \rightarrow B.CD, \{S(t_1,t_2)\}) = \{S(t_1,t_2 \mid A \rightarrow B.CD\}$

   is a *grammatical theory*: a dotted rule that has found constituent B (deriving a sequence of words $w_i,w_{i+1},...w_j$ with acoustic segment scores $\{S(t_1,t_2)\}$), and looking to match a C.

Lattice parsing can now be described in terms of the following set of operations (compare this with the summary of text parsing above on page 23):

1. Input is a lattice of *alternate* word theories.
2. Entries in the chart are *grammatical theories* of the form:

   $(A \rightarrow B.CD, \{S(t_1,t_2)\})$

3. Dot movement:

$$\begin{aligned} \text{if } A \rightarrow B.\ C\ D,\ \{S(t_1,\ t_2)\} &\Rightarrow \text{input}[i,\ j] \\ \&\ C \rightarrow E\ F.,\ \{S(t_3,\ t_4)\} &\Rightarrow \text{input}[j,\ k] \\ \text{then } A \rightarrow B\ C.\ D,\ \{S(t_5,\ t_6)\} &\Rightarrow \text{input}[i,\ k] \end{aligned}$$

   where $\{S(t_5,t_6)\}$ is computed by a DP solution from $\{S(t_1,t_2)\}$ and $\{S(t_3,t_4)\}$:

   $S(t_5,t_6 \mid A \rightarrow BC.D) = min_\tau\ S(t_5,\tau \mid A \rightarrow B.CD) + S(\tau,t_6 \mid C \rightarrow EF)$

4. Grammatical theories are stored in a chart:

   $\mathbf{dr}[i,k] = \{(A \rightarrow B.C, \{S(t_1,t_2)\}) \mid B \Rightarrow input[i,k]\}$

The output the lattice parser is a set of complete parses spanning the entire utterance with associated acoustic likelihood scores $\{S(1,T \mid (START)) \Rightarrow \gamma.)\}$.

This chapter is organized as follows: in Section 4.1, we discuss the speech component of our speech understanding system that is used to compute the acoustic scores for the words in the vocabulary; in Section 4.2, we give details of our integration of speech and syntax; in Section 4.3, we describe how we currently incorporate semantics to find the best interpretation of the input; in Section 4.4 we discuss some of the system implementation issues in building integrated BBN Spoken Language System (SLS); and finally in Section 4.5, we discuss the performance of the overall system.

4.1 Speech

In the architecture we have chosen for speech and natural language integration, there are two processes that are part of the recognition procedure: (1) speech acoustic scoring, which uses only speech recognition technology; and (2) language model scoring, which uses both acoustic and linguistic information. Therefore, one needs to ensure that sufficient computing is performed in the first stage and enough information is saved so that optimality is preserved in the later stages of processing. To achieve this goal, our lattice computation algorithm attempts to compute acoustic likelihood scores for all words in the vocabulary V for all time intervals i and j. We define the acoustic score of a word w to be the logarithm of the conditional probability:

$$S(i,j \mid w)=Log(prob(x(i),x(i+1),...x(j) \mid w))$$

where (prob(x(i), x(i+1), ... x(j) | w)) is the likelihood that the terminal or word w produced the observed input acoustic data between times i and j. The acoustic data is typically a sequence of analyzed and vector-quantized (VQ) input spectra sampled every 10 millisecond [12]. We model the input speech at the phonetic level using robust context-dependent Hidden Markov Models (HMM) of the phoneme [44]. The acoustic models for the words in the vocabulary are derived from the concatenation of these context-dependent phonetic HMMs.

Using these acoustic models of the word, one can compute the acoustic scores for each word on the input utterance using a nonlinear time alignment—or Dynamic Time Warping (DTW)—procedure. A computationally efficient method is to use the trellis algorithm [27]. Our first implementation of a DTW algorithm used backward (in time) trellis computation to compute all scores $\{Prob(i,j|W):0<i<j\leq T\}$ in a single pass for a fixed time j, as shown in Figure 4-1.

In this version of the algorithm the index i only needs to range from j minus 1 down to j minus the maximum duration for the word, which in our system is based on the number of phonemes in the word. We further improved on the computational efficiency by switching the order of the W loop and the i loop. This allows us to do time-synchronous (beam) pruning similar to that described in [44] among all words that ended at time j. This pruning algorithm compares words that end at the same time j and eliminates those that score poorly acoustically relative to the best scoring word for all time $i,i<j$. It has given us a factor of two or more reduction in the acoustic computation. The improved algorithm is shown in Figure 4-2.

The computational complexity of this backward DTW algorithm is proportional to $2\times J^2\times T$, where J is the

```
;; For all ending times j
for j = 1, T   do

    ;; For all words in the vocabulary
    for W in {W}   do

        ;; For all beginning times i compute Prob(i,j|W)
        for i = j-1, Max(0, j-MaxDur[W]) by -1 do

            Perform within-word DTW and compute Prob(i,j|W)
```

where DTW is performed using the trellis algorithm.

Figure 4-1: Dynamic Time Warping (DTW) algorithm 1

```
;; For all ending times j
for j = 1, T   do

    ;; For all beginning times i compute Prob(i,j|W)
    for i = j-1, 1 by -1 do

        ;; For all words in the vocabulary
        for W in {W}   do
        if (i < j-MaxDur[W])
           quit

           else

             Perform within-word DTW and compute Prob(i,j|W)
             Also keep track of maximum score across words and
             perform time-synchronous pruning.
```

Figure 4-2: Dynamic Time Warping (DTW) algorithm 2

maximum duration of a word and T is the length of input utterance in frames. The result of performing this acoustic computation is the word lattice

$\{S(i,j|w)\}; 0 \le i < j \le T, \forall w \in V$

which is then used as input to the lattice parser.

4.2 Integration of Speech and Syntax

Previously, we argued for the need to integrate speech and natural language, as integration is essential for optimizing the performance of a speech understanding system. In this section, we describe our efforts in integrating speech and syntax, and use this as the basis for incorporating other natural language components. Before describing the algorithms for integration, we will first review the syntactic component of our natural language system (for a detailed discussion, see Chapter 2). As described previously, our system uses a unification grammar for representing the syntax. A unification grammar is essentially a context-free grammar (CFG) augmented with variables. The algorithm used for performing syntactic analysis operates word-synchronously, left-to-right and bottom-up and computes all possible parses of the input. It is similar to the CKY parsing algorithm that appears in the literature on context-free grammars. It starts at the terminals and iteratively derives larger grammatical constituents spanning the smaller ones that have already been found. Building a larger constituent from subconstituents involves *unification* (see Chapter 2)—the process of matching terms (made of complex expressions) in the grammar, requiring recursive computation, which is a compute-intensive and memory-intensive process. This algorithm is shown to have a computational complexity proportional to N^3, where N is the length of the input text.

We describe three parsing algorithms that have been implemented for integrating speech and syntax. All are natural extensions of the text parser. The first is a time-synchronous parsing algorithm that operates at the resolution of the frame (10 millisecond). However, this algorithm is extremely complex computationally, rendering it impractical. Our second implementation is a word-synchronous parser more similar to the text parser. It takes advantage of the redundancy across time frames by combining similar constituents that occur across different time intervals into a single constituent and parsing with only this single contituent. A computational saving of two orders of magnitude has been realized using this algorithm. The third algorithm is also word synchronous, but uses a top-down prediction mechanism to prune extraneous constituents from the parse.

We describe the three speech parsers below. In subsequent sections, all discussions pertain only to the word-synchronous parser, which is our current implementation.

4.2.1 The Time-Synchronous Speech Parser

Figure 4-3 presents the algorithm for the time-synchronous lattice parser.

As can be seen, this lattice parser is similar in many respects to the text parser. (Compare it to the algorithm in Figure 2-2.) The parser builds the table `dr[i,j]` starting from time j=1 and marches left to right, filling the table with valid grammatical constituents. What is distinctly different is that i and j range over time/frame positions within the utterance rather than over word positions, and that each grammatical constituent has been augumented with an acoustic likelihood score `S[i,j]`. The process of parsing involves matching terms to derive larger constituents as well as combining the acoustic scores from substituents to arrive at a new acoustic score. The major drawback of this algorithm is its computation and storage complexity. Since the parsing algorithm runs in time

```
;;; First compute the word lattice
   for all terminals W
       Compute acoustic likelihood score S(i,k|W), i < k < T
       using DTW
;;; For all ending time
for k = 1 to T
   ;;; For all starting times
   for i = k-1 to 0 by -1
       ;;; Compute chart entries for time interval <i,k>
       dr[i,k] =
           {(A → W. ε, S[i,k|W]) | W ∈ input [i,k]} ∪
             {(A → α B. β, S[i,j|A] + S[j,k|B] |
                    (A → α. B β , S[i,j|A]) ∈ dr[i,j]
                    & (B → γ., S[j,k|B]) ∈ dr[j,k]}
       MaxScore[i,k|A] = Max S[i,k|A]   for all (A → δ.) ∈ dr[i,k]
       Traceback[i,k|A] = (A → δ.) = Arg(MaxScore[i,k|A])
       i < j < k
```

where

`dr[i,j]`	= dotted rule table consisting of grammatical constituents spanning time interval *i,j*|*i*<*j*
A, **B**	= lefthand sides of rules in the grammar
α, β, γ, δ, ε	= symbols deriving arbitrary number of terminals
`MaxScore[i,k|A]`	= the best scoring grammatical constituent spanning `input[i,k]` with **A** as the lefthand side
`Traceback[i,k|A]`	= the lefthand side of **A** → δ. with the best score for `input[i,k]`

Figure 4-3: Time-synchronous Lattice Parsing Algorithm

proportional to the length of the input, and the length in this case is T, the number of frames in the speech utterance, this algorithm would run in time proportional to 300^3 (assuming an utterance is 3 seconds long)! Also, as stated, the algorithm only keeps the single best scoring parse for a time interval <i,j> and throws away all others. Alternate syntactical interpretation of the same input are explicitly discarded—making subsequent application of semantics impossible. We propose a superior parsing algorithm—the word-synchronous parser—described below.

4.2.2 The Word-Synchronous Speech Parser

In the time-synchronous parser, the entries in `dr[i, j]` contain theories of the form:

`(A → B. C D, S[i, j])`

where B spans `input[i, j]` with acoustic score `S[i, j]`. In all likelihood, `dr[i, j+1]` would include

`(A → B. C D, S[i, j+1])`

where the same dotted rule A → B. C D is being computed twice.

As unification is expensive computationally, much could be gained by removing the redundant representations in the parse table, and thereby minimizing the number of unifications computed. One way to achieve this is by grouping these two theories into a single theory,

`(A → B. C D , {S[i, j], S[i, j+1]})`

In fact, one could collect all neighboring theories into a single theory.

(A → B. C D , {S[i,j]})

where {S[i,j]} is the set of scores, and associated with each is a starting time $i, i \in$ <Imin,Imax>, and an ending time $j, j \in$ <Jmin,Jmax>. In effect, this groups a contiguous region of the input utterance into word units, and then applies the parser at the word level to find the best syntactic parse. This is the word-synchronous lattice parser. Figure 4-4 presents the algorithm for the word-synchronous parser.

```
First for all terminals W
    compute (W, {S[t1,t2]})

;;; For ending positions k (in words)
for k = 1 to N

    ;;; For starting positions i (in words)
    for i = k-1 to 0 by -1

        ;;; Compute chart entries for word positions <i,k>
        dr[i,k] =

                (if i+1 = k

                    {(A → W. α , {S[t1,t2]}) | W ∈ input [i,k]}

                 else

                    {(A → α B. β , {S[t5,t6]}) |

                     (A → α. B β , {S[t1,t2]}) ∈ dr[i,j]

                      & (B → γ. , {S[t3,t4]}) ∈ dr[j,k]}

                     i < j < k
                )

                ∪

                {(A → B. α , {S[t5,t6]}) | (B → γ. , {S[t5,t6]}) ∈ dr[i,k]}

                ∀ (A → B α) ∈ P

    where
            t1 ∈ <T1min,T1max>, t2 ∈ <T2min,T2max>
            t3 ∈ <T3min,T3max>, t4 ∈ <T4min,T4max>
            t5 ∈ <T5min,T5max>, t6 ∈ <T6min,T6max>
    and
            {S[t5,t6]} = {S[t1,t2]} ## {S[t3,t4]} ≠ {}
```

Figure 4-4: Word-synchronous Parsing Algorithm

The operator ## is defined as the concatenation of two sets of acoustic scores {S[t1,t2]} and {S[t3,t4]} to derive a new set of scores spanning the intervals of the two sets, using the following DP algorithm:

```
For ta ∈ <T1min,T1max>
    For tb ∈ <T4min,T4max>
        S[ta,tb] =  Max(S[ta,ti] + S[ti+1,tb])
                     ti

                    where ti ∈ <T2min,T2max> ∩ <T3min,t3max>
```

While the computational complexity of the time-synchronous algorithm was T^3, that of the word-synchronous algorithm is N^3, where N is now the estimated number of words in the speech signal; part of the task of the parser is to determine the true word sequence from the signal, and therefore the value of N. The implementation of the word-synchronous lattice parser is described in detail in Section 4.4.

4.2.3 Bottom-up Lattice Parsing with Prediction

The word synchronous lattice parser, like the text parser, is essentially an all paths parser—it forms partial parse hypothesis based on local bottom-up information only, without taking global contextual information into account. For example,

$B \stackrel{*}{\Rightarrow} many\ ships \in \mathbf{dr}[j,k]$

may be a partial constituent in the chart, but there may be no dotted rule of the form

$A \rightarrow \alpha.B\ \beta \in dr[i,j]$

that needs a B following its dot. This could result in a large number of extraneous constituents in the chart that do not contribute to a complete parse of a sentence. One solution is to use prediction to enforce global contextual constraints on what can be formed bottom-up, resulting in a chart significantly smaller in size. We first describe prediction for the text parser, and then extend it to the case of lattice parsing.

The idea for using prediction is the following. In the bottom-up (text) parser, entries in the chart have the following property:

$$\mathbf{dr}[i, k] = \{\ A \rightarrow B\ .\ C \mid B \stackrel{*}{\Rightarrow} \text{input}\ [i, k]\ \} \tag{1}$$

When *left-to-right top-down prediction* is used, the entries in the chart are more constrained:

$$\mathbf{dr}[i, k] = \{\ A \rightarrow B\ .\ C \mid B \stackrel{*}{\Rightarrow} \text{input}\ [i, k]\} \\ \&\ (\text{STAR1}) \stackrel{*}{\Rightarrow} \text{input}\ [0, i]\ A\ \gamma \tag{2}$$

As shown in equations 1 and 2, the grammatical constituents in the chart have to satisfy an additional constraint that they have to be *predicted* to be derived by the complete utterance constituent ((*START*)) from the beginning of the input. This can be a very powerful constraint in eliminating extraneous partial parses from the chart.

The prediction algorithm is formulated as follows. We assume that the following prediction table Φ has been precomputed.

$\Phi = \{[B\ C] \cdot B \stackrel{*}{\Rightarrow} C\beta\}$

This is the set of pairs $[B\ C]$ such that given B as the left hand symbol of a rule, C can be *derived* as the first symbol on the right hand side. The algorithm for computing Φ is given in by the following multi-step procedure:

1. Initialize P_1 to be the set of pairs $[A\ B]$ such that $A \rightarrow B\beta$ is a production rule, and initialize n to 1

$$P_1 \leftarrow \{[A\ B] \mid A \rightarrow B\beta \in \mathbf{P}\}$$
$$n \leftarrow 1$$

2. $P_{n+1} \leftarrow LinkProd(P_n\ P_1)$ where $\mathbf{Z} = LinkProd(\mathbf{X}\ \mathbf{Y})$ is defined as:

   a. Initialize $\mathbf{Z}$ to be the empty set
   $\mathbf{Z} \leftarrow \{\}$

   b. Take symbolic cross products of sets $\mathbf{X}$ and $\mathbf{Y}$ by performing unification of the *right hand side* of every X with the *left-hand side* of every Y

   c.
$$\forall (X = [A\ B]) \in \mathbf{X}$$
$$\forall (Y = [B'\ C]) \in \mathbf{Y}$$
$$\textbf{if } Unify(B\ B')$$
$$\textbf{then } \mathbf{Z} \leftarrow \mathbf{Z} \cup [A\ C]$$

3. **if** $P_{n+1} \neq \{\}$
   then $n \leftarrow n+1$
   goto step 2

4. Φ is the union of $P_n, n>0$

$$\Phi \leftarrow \cup P_n$$

At each word position j during parsing, one would like to *predict* what symbols B can come next, denoted by $\mathbf{pred}[j]=\{B \mid (START) \rightarrow input[0,j]B\gamma\}$

The set of predictions **pred[j]** is computed by the following two steps:

1. First, compute the set of *directly expected* symbols $\mathbf{D}$:

$$\mathbf{D}=\{B \mid (A \rightarrow \alpha.B\gamma) \in \mathbf{dr}[i,j], \forall i<j\}$$

2. Then, compute the set of *indirectly expected* symbols $\mathbf{E}$ from $\mathbf{D}$ using the prediction table Φ, i.e.:

$$E=\{C \mid B \in \mathbf{D}\ \&\ B \stackrel{*}{\Rightarrow} C\beta\}$$

3. The set of predicted symbols is the union $\mathbf{pred}[j]=\mathbf{D} \cup \mathbf{E}$

4. The set $\mathbf{pred}[j]$ is then used to *filter* out the entries in $\mathbf{dr}[j,k]$, $\forall\ k>j$ to reduce to the set:

$$\mathbf{dr}[j,k]=\{A \rightarrow B.C \mid B \stackrel{*}{\Rightarrow} input[j,k] \&\ A \in \mathbf{pred}[j]\}$$

The use of prediction works by induction. By ensuring that partial constituents are consistent with the $(START)$ symbol starting at sentence initial position (word index 0), all subsequent entries in the chart at positions $j, j>0$, will be guaranteed to be consistent with $(START)$.

As in text parsing, prediction can be used in parsing the lattice. Extending equation 1 to the case of lattice input, entries in the chart in bottom-up parsing have the following property:

$$\mathbf{dr}[i, k] = \{ (A \rightarrow B\ .\ C, \{S(t_1, t_2)\}) \mid B \stackrel{*}{\Rightarrow} \text{input}[i,k] \} \quad (3)$$

Using *left-to-right top-down prediction* (from equation 2)

$$\mathbf{dr}[i,k] = \{\ (A \rightarrow B\ .\ C,\ \{S(t_1, t_2)\})\ |\ B \stackrel{*}{\Rightarrow} input[i,k]\ \} \\ \&\ (START) \stackrel{*}{\Rightarrow} input[0,i]\ A\ \gamma \qquad (4)$$

Again, as in text parsing, the grammatical theories in the chart have to satisfy an additional constraint that they have to be *predicted* to follow a complete sentential constituent (*START*) from the beginning of the utterance. This is a very powerful constraint, especially when the input is a lattice, where most of the constituents formed will be nonsensical. As in text parsing, we can compute predictions (*pred*[*j*] at position *j* in the input and then use them to *filter* out the entries in **dr**[*j*, *k*],$\forall k>j$:

$$\mathbf{dr}[j,k]=\{A\rightarrow B.C,\{S(t_1,t_2)\}\ |\ B \stackrel{*}{\Rightarrow} input[j,k] \\ \&\ A\ \in pred[j]\}$$

4.2.4 Merging Grammatical Theories

Grammatical theories in the chart belonging to the same chart entry $dr[i,j]$ and labeled with the same non-terminal symbol are candidates for *merging*; merging is necessary to avoid exponential growth in ambiguity, especially when the input is speech. The technique of merging works as follows. Non-terminal symbols in the chart (representing top nodes of subtrees) are combined into a single structure and subsequently treated by higher level structures as equivalent to a single node. Completed grammatical constituents (dotted rules with the dot at the end, or symbols) in the chart that have the same left hand side *A*, e.g.

$A\rightarrow \gamma.A \rightarrow \zeta.,$

can be merged to form a single symbol *A*,

$(A\ |\ A \rightarrow \gamma.A \rightarrow \zeta)$

with two different parse trees. Instead of having separate entries in the chart, we can represent them with one entry, but still keep distinct parse trees for the various different ways of deriving *A*. Using this theory merging scheme, one can reduce computation dramatically, yet be able to keep track all possible parses on the input lattice.

4.2.5 Computational Complexity

The computational complexity of the bottom-up parsing algorithm is at least proportional to RN^3S^β, where *R* is the number of rules in CNF, *N* is the length of the utterance, *S* is the size of the lattice, and β is some exponential factor greater than one. Correspondingly for the predictive parsing algorithm, the complexity is proportional to $\alpha RN^\xi S^\kappa + \beta C_p$, where α and β are proportionality constants, ξ is some constant less than 3, and C_p is the cost of computing the predictions. In practice, the computation associated with prediction can be a significant part of the total computation. However, predictions in general reduce the number of entries in the chart dramatically, and therefore can reduce the complexity of the algorithm to much less than cubed ($\xi<<3$), which is very desirable. Often times the use of prediction is essential to making the computation tractable.

4.3 Integrating Semantics

Our initial strategy for applying semantic interpretation to the speech parser is similar to that in the text parser, i.e., after syntactic analysis has been completed. To allow for this without incurring the cost of repeating the same parsing computation over and over again for parsed constituents that are the same but with different parse trees, a scheme for representing the entries in the parse table **dr** was devised. It is as follows. The entries in the parse table **dr[i,j]** for a particular *i* and *j* is partitioned into equivalence classes. Within each equivalence class are theories having been parsed to the same grammatical expression (Gexpr), but with possibly different parse trees corresponding to different ways of parsing a particular word sequence in the lattice as well as to those that correspond to parses of different word sequences which happened to have resulted in the same parsed constituent. Each equivalence class is headed by a representative Gexpr template on which unification matching is performed (therefore, only a single unification match is performed per class); however, tree building and speech concatenation computation (unique to each member within the class) are done separately for each member within the class.

To illustrate,

```
If (A → B. C D , {S[t1,t2]}) ∈ dr[i,j]

  & {<C>: (C → E F. {S[t3,t4]})
          (C → E' F'. {S[t3',t4']})
          (C → E'' F''. {S[t3'',t4'']})
    } ∈ dr [j,k]

    Then {<A → B C. D>: (A → B C. D {S[t5,t6]})
                        (A → B C. D {S[t5',t6']})
                        (A → B C. D {S[t5'',t6'']})
         } ∈ dr[i,k]
```

where

<C> = representative Gexpr template for the equivalence class C
<A → B C. D> = resulting representative dotted rule.

By representating the parse table in this way, the parser is able to find all parses of the input without replicating the same work, which in the worst case could have exponentiating effects on computation.

Finally, semantic interpretation is performed by first finding the set of complete constituents (i.e., all entries of the form **((START) {SCORES})** in the parse table (i.e., **dr[0,i]** *i*>0) that span the entire utterance <1,*T*>, and then by finding the best scoring one of those which is also semantically meaningful. This gives us the single best answer that is optimal with respect to speech, syntax, and semantics.

4.4 System Implementation

Many practical issues were encountered in the implementation of our SLS system on the Symbolics Lisp Machine (LISPM). In this section, a representative subset of these issues will be highlighted and methods for handling them discussed. The issues and discussions are relevant only to the current implementation—the word-synchronous lattice parser.

4.4.1 Silence Handling

The word lattice computed by the speech component contains words that are in the speech lexicon, including the word **SILENCE**, representing intervals in the utterance where the model for **SILENCE** matched well against the input speech (as silence is located at utterance beginnings, utterance endings, at the beginning of plosives, actual pauses in speech, etc). Since silence, or pause, is not explicitly handled in the natural language grammar (in the future, we may include pause detections to help identify phrase boundaries), something must be done to eliminate these silences in the word lattice while maintaining its integrity. We handle this by merging silences into the neighboring words: all words that have silences as neighbors would also have another instance created that has its boundary extended to include silence, with the proper acoustic scores included. By incorporating this silence merging stage as a preprocess to the parser, we have eliminated the need to modify the grammar/parser so it could handle silence explicitly, while ensuring that all of the input signal is accounted for.

4.4.2 Search Strategies

Search strategy design is by far the most important task in designing and implementing a system dealing with a large search space such as the one we are building now, as efficient search strategies can mean orders of magnitude reduction in both computation and memory requirements. In this section we describe some of the methods employed in our system that are used to prune down the search space.

4.4.2.1 Conditions for Search Termination

As mentioned previously, the computational complexity of the word-synchronous parser is proportional to N^3, where N is the estimated/computed length of an utterance in words. To minimize unnecessary computation, one needs to detect N as early as possible and terminate search (the bottom up parser could go on for a long time beyond the actual number of words in the input). The algorithm that we have devised is as follows: before each parse, we compute the best scoring word sequence from the word lattice without using any grammar constraints (i.e., all words can follow any word). The resulting acoustic score of this answer is used as an upper bound on the score of the best scoring word sequence allowed by the grammar. The condition for terminating search is satisfied when we are at some position k in the parsing algorithm where a complete grammatical constituent (with the symbol **(START)**) spanning the entire utterance is found in `dr[0,k-1]` with a score within a threshold of the upper bound score. The

reason for choosing to look at `dr[0,k-1]` rather than `dr[0,k]` is a subtle but well-motivated one: a valid complete constituent may have short function words (such as "a" or "the") deleted from it and still score well enough to satisfy the search termination criterion. In other words, we always want to compute one more word given that we think that we have found the "correct" word sequence. By hedging against single word deletions, we are also relying on the assumption that more word deletions will deteriorate the overall score to the level where the threshold test (against the upper bound score) can no longer be satisfied. Finally, the best answer is computed by searching over all theories `((START) {SCORES})` in `dr[0,i]`, $i>0$ that span the utterance from time 1 to T and finding the theory with the best score.

4.4.2.2 Reduction In Time Resolution

A simple scheme to reduce the computation is to down-sample the backward DTW in computing the word lattice to compute at every T frame (for example, skip every other frame). This would reduce the speech lattice computation by the same factor, and would reduce the score concatenation operation (`##`) in the parsing by the same factor squared. This is a simple and straightforward (and well understood) method for cutting down on computational load with minimal loss in performance, and we, in fact, make this the default mode of operation for our system. Currently, we use a time resolution of two (skip every other frame) in running our system.

4.4.2.3 Word Lattice Pruning

As described earlier, the speech component computes a very dense word lattice which potentially would include all words in the lexicon with scores between every time interval $<i,j>$. The motivation behind using a such a dense word lattice in the parser is to be sure with probability close to one that the correct words would be included at the right place in the lattice. However, to consider such a lattice in its entirety would not only be a waste of effort since most of words are in fact just noise (as our models of the word are probabilistic), but would be computationally impractical for our bottom-up parser. An integral part of designing the lattice parsing algorithm is to come up with ways of reducing the size of the lattice and yet ensuring that the correct word sequence is present.

A straightforward approach is to prune down the word lattice, keeping only those words that score reasonably well (all those that have an average score per frame > δ, where δ is a predetermined threshold) and ignoring all those that scored poorly acoustically. The key, then, is to determine δ so that it would result in a high hit rate for the correct words while minimizing the detection of irrelevant words. In practice, we found it very difficult if not impossible to find such a δ that would fullfill these two conflicting requirements, as words have widely varying acoustic scores not only among themselves, but across contexts and environments. In fact, one can imagine that in the worst case, a distinct threshold would be needed for every word in every context. Fortunately, from empirical evidence as well as from our knowledge of acoustics, we infer that the average acoustic scores of words in general tend to fall into two broad classes: short words (most function words, such as "a", "the", "of", that have two syllables or less), and long words (such as proper nouns like "Frederick" and "Westpac"). The short words, since they are short in duration and are often reduced in spoken utterances, tend to score poorly acoustically; whereas the long words are often spoken more carefully, and would have consistently higher acoustic scores. This suggested the use of dual thresholds—one for short words and the other for long words. In fact, we have generalized this to an

arbitrary number of classes, with short words on one end of the spectrum and long words on the other, with other word classes in between, with the guiding principle that words with fewer syllables would tend to be more variable acoustically, and therefore would need smaller thresholds, and vice versa. Currently, our system employs four levels of word level pruning. And in practice, we have found that this multi-level pruning reduces the size of the lattice (as measured by the number of words, each with a set of boundaries with associated acoustic scores) by at least a factor of two. This reduces the parsing computation dramatically as the lattice size has an exponentiating effect on the size of the parse table.

4.4.2.4 Pruning During Speech Parsing

One strategy for pruning the search space during parsing is the time-synchronous beam search used in the BYBLOS speech recognition system. However, the beam search used in the BYBLOS system always compares theories spanning the same time interval from time 1 (beginning of utterance) to current time t, whereas the bottom-up parser used here generates theories that can span arbitrary time intervals $<t1,t2>$. We have modified the beam search to work for our parsing strategy as follows. During parsing, the maximum score spanning a particular time interval $<t1,t2>$ (for all time intervals) is computed, and theories that span $<t1,t2>$ are only kept if their score is within a threshold (the beam) of the maximum; all others are eliminated. This method is less effective when theories are short (spanning short time intervals), and much more effective as theories become long in duration. Empirically, we've found this pruning strategy reduces computation significantly over no pruning.

4.5 System Performance

In this section, we present results for the BBN Spoken Language System on the standard DARPA 1000-Word Resource Management speech database [40], with 600 sentences (about 30 minutes) of training speech to train the acoustic models for each speaker. For these experiments, speech was sampled at 20 kHz, and 14 Mel-Frequency cepstral coeffients (MFCC), their derivatives (DMFCC), plus power (R0) and derivative of power (DR0) were computed for each 10 ms, using a 20 ms analysis window. Three separate 8-bit codebooks were created for each of the three sets of parameters using K-means vector quantization (VQ). The experiments were conducted using the multi-codebook paradigm in the HMM models, where the output of vector quantizer, which consists of a vector of 3 VQ codewords per 10 ms frame, is used as the input observation sequence to the HMM.

For the purpose of making computation tractable, we applied the lattice pruning techniques described above to a full word lattice to reduce the average lattice size from over 2000 word theories to about 60^{38}. At this lattice size, the probability of having the correct word sequence in the lattice is about 98%, which places an upperbound on subsequent system performance using the language models.

[38] 60 word theories corresponds to about 4000 micro acoustic scores.

Language Model	Perplexity	% Word Error	% Sentence Error
None	1000	15.45	71.3
Word Pair	60	3.9	26.0
Syntax	700	7.5	38.0
Syntax+Semantics	NA	6.9	36.4

Figure 4-5: Recognition Performance of the BBN Spoken Language System

Figure 4-5 shows the results averaged across 7 speakers, using a total of 109 utterances, under 4 grammar conditions. As shown, the grammars tested include: 1) *no* grammmar: all word sequences are possible; 2) the *word pair* grammar, containing all pairs of words occuring in the set of sentences that was used to define the database; 3) the *syntactic grammar* alone; and 4) semantic interpretation for *a posteriori* filtering on the output of lattice parsing.

Note that the performance using the *syntactic* language model is 7.5% error. At a perplexity of 700, its performance should be closer to the *no grammar* case, which has a perplexity of 1000 and an error rate of about 15%. We hypothesize that perplexity alone is not adequate to predict the quality of a language model. In order to be more precise, one needs to look at *acoustic perplexity*: a measure of how well a language model can selectively and appropriately limit acoustic confusability. A linguistically motivated language model seems to do just that—at least in this limited experiment. Also, surprisingly, using semantics gave insignificant improvement in the overall performance. One possible explanation for this is that semantics gets to filter only a small number of the sentences accepted by syntax. Out of the sentences which receive semantic interpretations, syntax alone determined the correct sentence better than 60 percent of the time, leaving only about 20 sentences in which the semantics has a chance to correct the error. Unfortunately, of these errorful answers, most were semantically meaningful, although there were some exceptions. Pragmatic information may be a higher level knowledge source to constrain the possible word sequences, and therefore improve performance.

5. Publications and Presentations

5.1 Publications

Ayuso, D., Y. Chow, A. Haas, R. Ingria, S. Roucos, R. Scha, and D. Stallard (1988) *Integration of Speech and Natural Language: Interim Report*, Report No. 6813, BBN Laboratories Incorporated, Cambridge, Massachusetts.

de Bruin, Jos and Remko Scha (1988) "The Interpretation of Relational Nouns", *26th Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference*, Association for Computational Linguistics, Morristown, NJ, pp. 25--32.

Haas, Andrew (1987) "Parallel Parsing for Unification Grammars", *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 615--618.

Haas, Andrew (to appear) "A New Parsing Algorithm for Unification Grammar", *Computational Linguistics*.

Ingria, Robert J. P. (1988) "Natural Language Processing: Where It's Been and Where It Might Be Going", *Proceedings of the IV-th Conference: Computer Processing of Language Data*, Jozef Stefan Institute, Ljubljana, Yugoslavia, pp. 59--73.

Ingria, Robert J. P. (1989) "Not All Utterances Are Sentences", *Proceedings of WECOL 18: The Western Conference on Linguistics*.

Ingria, Robert J. P. (to appear) "Simulation of Language Understanding: Lexical Recognition", chapter 30 in the *Computational Linguistics* volume of the Handbuecher zur Sprach- and Kommunikationswissenschaft series, Walter de Gruyter and Co., Berlin, pp. 337--350.

Scha, Remko (1988) "Natural Language Interface Systems", in M. Helander, ed., *Handbook of Human-Computer Interaction*, North-Holland, Amsterdam, pp. 941--956.

Scha, Remko and Livia Polanyi (1988) "An Augmented Context Free Grammar for Discourse", *Proceedings of the 12th International Conference on Computational Linguistics, 22-27 August 1988, Budapest, Hungary*.

Scha, Remko and David Stallard (1988) "Multi-Level Plurals and Distributivity", *26th Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference*, Association for Computational Linguistics, Morristown, NJ, pp. 17--24.

5.2 Abstracts Accepted

Haas, Andrew (1989) "A Generalization of the Offline-Parsable Grammars", 27th Annual Meeting of the Association for Computational Linguistics, University of British Columbia, 26--29 June, 1989.

Ingria, Robert J. P. and David Stallard (1989) "A Computational Mechanism for Pronominal Reference", 27th Annual Meeting of the Association for Computational Linguistics, University of British Columbia, 26--29 June, 1989.

5.3 Presentations

Ingria, Robert J. P. "Adjectives, Nominals, and the Status of Arguments", AAAI Workshop on Theoretical and Computational Issues in Lexical Semantics, Brandeis University, Waltham, Massachusetts, April 22, 1988.

Ingria, Robert J. P. "The Grammar of the BBN Spoken Language System", Boston University Computer Science department Fall 1988 Colloquium Series, Boston University, Boston, Massachusetts, January 25, 1989.

Ingria, Robert J. P. and James Pustevosky "Active Objects in Syntax, Semantics, and Parsing", Parsing Seminar, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 7, 1988

Stallard, David "The Spoken Language System", Boston University Computer Science department Fall 1988 Colloquium Series, Boston University, Boston, Massachusetts, November 16, 1988.

References

[1] Alexander, D. and W.J. Kunz.
Some Classes of Verbs in English.
distributed by Indiana University Linguistics Club. Bloomington, Indiana, 1964.
Linguistics Research Project, Indiana University, F. W. Householder, Jr., Principal Investigator.

[2] Aoun, Yousef and Dominique Sportiche.
On the Formal Theory of Government.
The Linguistic Review 2:211--236, 1983.

[3] Bobrow, R.J.
The RUS System.
Technical Report Section of BBN Report No. 3878, Bolt Beranek and Newman Inc., 1978.

[4] Bresnan, Joan.
Contraction and the Transformational Cycle in English.
distributed by the Indiana University Linguistics Club. Bloomington, Indiana, 1978.
Originally written in 1971.

[5] Bresnan, Joan W.
The Mental Representation of Grammatical Relations.
MIT Press, Cambridge, Massassuchetts, 1982.

[6] Bridgeman, Loraine I., Dale Dillinger, Constance Higgins, P. David Seaman, Floyd A. Shank.
More Classes of Verbs.
distributed by Indiana University Linguistics Club. Bloomington, Indiana, 1965.
Linguistics Research Project, Indiana University, F. W. Householder, Jr., Principal Investigator.

[7] W.J.H.J. Bronnenberg, H.C. Bunt, S.P.J. Landsbergen, R.J.H. Scha, W.J. Schoenmakers and E.P.C. van Utteren.
The Question Answering System PHLIQA1.
In L. Bolc (editor), *Natural Language Question Answering Systems.* Macmillan, 1980.

[8] Carlson, G.
Reference to Kinds in English.
PhD thesis, University of Massachusetts, 1977.

[9] Chomsky, Noam.
Syntactic Structures.
Mouton, The Hague, 1957.

[10] Chomsky, Noam.
On Binding.
Linguistic Inquiry 11(1):1--46, 1980.

[11] Chomsky, Noam.
Lectures on Government and Binding.
FORIS PUBLICATIONS, Dordrecht-Holland/Cinnaminson-U.S.A., 1981.

[12] Y.L. Chow, M.O. Dunham, O.A. Kimball, M.A. Krasner, G.F. Kubala, J. Makhoul, P.J. Price, S. Roucos, and R.M. Schwartz.
BYBLOS: The BBN Continuous Speech Recognition System.
In *International Conference on Acoustics, Speech, and Signal Processing*, pages 89-93. IEEE, Dallas, Texas, April, 1987.

[13] DeBruin, Jos and Scha, Remko J.H.
The Interpretation of Relational Nouns.
In *Proceedings of the 26th Annual Meeting of the ACL*, pages 25--32. Association for Computational Linguistics, June, 1988.

[14] Gazdar, Gerald, Ewan Klein, Geoffrey Pullum, Ivan Sag.
Generalized Phrase Structure Grammar.
Harvard University Press, Cambridge, Massachusetts, 1985.

[15] Graham, Susan L., Michael A. Harrison, and Walter L. Ruzzo.
An Improved Context-free Recognizer.
ACM Transactions on Programming Languages and Systems 2(3):415-461, 1980.

[16] Haas, Andrew.
Parallel Parsing for Unification Grammar.
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 615-618. IJCAI, Milan, Italy, August, 1987.

[17] Haas, Andrew.
A Parsing Algorithm for Unification Grammar.
forthcoming.

[18] Harman, Gilbert.
Generative Grammars without Transformation Rules: A Defense of Phrase Structure.
Language 39:597-616, 1963.

[19] Hays, D. G.
Introduction to Computational Linguistics.
American Elsevier, New York, 1967.

[20] Heidorn, George E.
English as a Very High Level Language for Simulation.
SIGPLAN Notices 9(4):91-100, April, 1974.

[21] Heidorn, George E.
Automatic programming through natural dialogue: a survey.
IBM Journal of Research and Development 20(4):302-313, July, 1976.

[22] Hoffman, Craig Ward.
Phrase Structure, Subcategorization, and Transformation in the English Verb Phrase.
PhD thesis, The University of Connecticut, Storrs, Connecticut, 1980.

[23] Ingria, Robert J.
A Summary of Verb Complement Types in English.
forthcoming.

[24] Ingria, Robert J.
Features in the BBN ACFG for Selected Verbs from the COBUILD Corpus.
forthcoming.

[25] Jackendoff, Ray.
Semantic Interpretation in Generative Grammar.
MIT Press, Cambridge, MA, 1972.

[26] Jaeggli, O.
Remarks on *to* Contraction.
Linguistic Inquiry 11:239-246, 1980.

[27] Frederick Jelinek.
Continuous Speech Recognition by Statistical Methods.
Proceedings of the IEEE 64(4):532-556, April, 1976.

[28] Joshi, Aravind K.
Factoring Recursion and Dependencies: An Aspect of Tree Adjoining Grammar (TAG) and a Comparison of Some Formal Properties of TAGs, GPSGs, PLGs, and LFGs.
In *Proceedings of the 21st Annual Meeting of the ACL*, pages 7--15. Association for Computational Linguistics, Massachusetts Institute of Technology, June, 1983.

[29] Kasami, T.
An efficient recognition and syntax analysis algorithm for context-free languages.
Technical Report Sci. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., 1965.

[30] Klima, Edward S.
Negation in English.
The Structure of Language: Readings in the Philosophy of Language.
Prentice-Hall, Englewood Cliffs, N. J., 1964.

[31] Lasnik, Howard.
Remarks on Coreference.
Linguistic Analysis 2(1):1--22, 1976.

[32] Marcus, Mitchell P.
A Theory of Syntactic Recognition for Natural Language.
The MIT Press, Cambridge, Massachusetts, 1980.

[33] Marcus, Mitchell P., Donald Hindle, and Margaret Fleck.
D-Theory: Talking about Talking about Trees.
In *Proceedings of the 21st Annual Meeting of the ACL*, pages 129--136. Association for Computational Linguistics, Massachusetts Institute of Technology, June, 1983.

[34] Marcus, Mitchell P.
Deterministic Parsing and Description Theory.
Linguistic Theory and Computer Applications.
Academic Press, 1987, pages 69--112.

[35] Montague, R.
The Proper Treatment of Quantification in Ordinary English.
In J. Hintakka, J.Moravcsik and P.Suppes (editors), *Approaches to Natural Language. Proceedings of the 1970 Stanford Workship on Grammar and Semantics*, pages 221-242. Dordrecht: D.Reidel, 1973.

[36] Pereira, Fernando C. N. and David H. D. Warren.
Definite Clause Grammars for Language Analysis---A Survey of the Formalism and a Comparison with Augmented Transition Networks.
Artificial Intelligence 13:231-278, 1980.

[37] Pereira, Fernando.
Extraposition Grammars.
American Journal of Computational Linguistics 7(4):243-256, 1981.

[38] Pollard, Carl and Ivan A. Sag.
Information-Based Syntax and Semantics.
Center for the Study of Language and Information, Stanford, CA, 1987.

[39] Postal, P. and G. K. Pullum.
The Contraction Debate.
Linguistic Inquiry 13:122-138, 1982.

[40] P. Price, W.M. Fisher, J. Bernstein and D.S. Pallett.
The DARPA 1000-Word Resource Management Database for Continuous Speech Recognition.
In *International Conference on Acoustics, Speech, and Signal Processing*, pages 651-654. IEEE, New York, New York, April, 1988.

[41] Procter, Paul et al, eds.
Longman Dictionary of Contemporary English.
Longman Group Limited, Harlow and London, 1978.

[42] Reinhart, Tanya.
The Syntactic Domain of Anaphora.
PhD thesis, Massachusetts Institute of Technology, 1976.

[43] Scha, Remko J.H. and Stallard, David G.
Multi-level Plurals and Distributivity.
In *Proceedings of the 26th Annual Meeting of the ACL*, pages 17--24. Association for Computational Linguistics, June, 1988.

[44] R. Schwartz, Y. Chow, O. Kimball, S. Roucos, M. Krasner, J. Makhoul.
Context-Dependent Modeling for Acoustic-Phonetic Recognition of Continuous Speech.
In *International Conference on Acoustics, Speech, and Signal Processing*, pages 1205-1208. IEEE, Tampa, Florida, March, 1985.

[45] Shieber, Stuart, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and Mabry Tyson.
The Formalism and Implementation of PATR-II.
Research on Interactive Acquisition and Use of Knowledge: Final Report SRI Project 1894.
SRI International, Menlo Park, California, 1983, pages 39--79.

[46] Shieber, Stuart M.
An Introduction to Unification-Based Approaches to Grammar.
CSLI (Center for the Study of Language and Information), Stanford University, Stanford, CA, 1986.

[47] Stallard, David.
A Manual for the Logical Language of the BBN Spoken Language System.
July, 1988.
Unpublished manuscript, BBN Systems and Technologies Corporation, Cambridge, Massachusetts.

[48] Sussman, Julie.
The Grapher.
Technical Report 6876, BBN Systems and Technology Corporation, Cambridge, Massachusetts, July, 1988.

[49] Vijay-Shankar, K. and Joshi, Aravind K.
Some Computational Properties of Tree Adjoining Grammars.
In *Proceedings of the 23rd Annual Meeting of the ACL*, pages 82--93. Association for Computational Linguistics, University of Chicago, July, 1985.

[50] Visser, F. Th.
An Historical Syntax of the English Language; Part One: Syntactical Units with One Verb.
E. J. Brill, Leiden, 1963.

[51] Visser, F. Th.
An Historical Syntax of the English Language; Part Two: Syntactical Units with One Verb (Continued).
E. J. Brill, Leiden, 1966.

[52] Visser, F. Th.
An Historical Syntax of the English Language; Part Three, First Half: Syntactical Units with Two Verbs.
E. J. Brill, Leiden, 1969.

[53] Visser, F. Th.
An Historical Syntax of the English Language; Part Three, Second Half: Syntactical Units with Two Verbs and with More Verbs.
E. J. Brill, Leiden, 1973.

[54] Younger, D. H.
Recognition and parsing of context-free languages in time ^{3}.
Information and Control 10(2):189-208, 1967.